# UNIVERSIDAD AUTONOMA DE MADRID

## ESCUELA POLITECNICA SUPERIOR




**Grado en Ingeniería Informática**

# TRABAJO FIN DE GRADO

## MODELADO DE INFORME DE CORRUPCIÓN CON RED NEURONAL TERNARIA


**Guillermo Jerez Benita**
**Tutor: David Renato Domínguez Carreta**


**MAYO 2017**

# MODELADO DE INFORME DE CORRUPCIÓN CON RED NEURONAL TERNARIA

**AUTOR: Guillermo Jerez Benita**
**TUTOR: David Renato Domínguez Carreta**

**Dpto. Ingeniería Informática**
**Escuela Politécnica Superior**
**Universidad Autónoma de Madrid**
**Mayo de 2017**

# Resumen (castellano)

En el presente trabajo se evalúa la posibilidad de emplear una red neuronal de tres estados para analizar datos sobre corrupción institucional, bajo la premisa de que un estado extra podría ayudar a separar los datos irrelevantes de aquellos que aportan contenido relevante dentro del conjunto de datos de entrada. Para implementar este algoritmo se ha usado un modelo de red neuronal Hopfield al que ha dotado de una matriz de pesos binaria y otra bimodal, construyendo de esta manera dos componentes a partir de las cuales dicha red neuronal adquiere como posibles estados de sus neuronas los valores -1, 0 y 1, donde 0 representa un estado de no-información o irrelevancia. Para evaluar el rendimiento de esta red de tres estados se ha tomado como parámetro más representativo el solapamiento entre un patrón de entrada y la salida de la red para dicho patrón, el cual ha sido separado análogamente entre las componentes de la red como solapamiento binario *m* y solapamiento bimodal *l*. En las pruebas realizadas en el marco de este trabajo se evalúa cómo se comportan los valores de estas componentes conforme el algoritmo va aprendiendo nuevos patrones, y si para alguno de los dos conjuntos de indicadores de corrupción evaluados se obtiene un valor de solapamiento bimodal superior al binario, lo que indicaría que la introducción de un nuevo estado tal y como se ha definido es efectivo a la hora de detectar contenido de relevancia frente a una red neuronal de las mismas características que solamente usara dos estados. Los resultados demuestran una potencial efectividad de este algoritmo para uno de los conjuntos de datos evaluados que, no obstante, hubiere de ser investigada en el futuro en mayor profundidad con conjuntos de datos más completos.



# Abstract (English)

This bachelor thesis evaluates the performance provided by a three-state neural network while analyzing institutional corruption indicators, on the premise that an extra state could improve the detection of relevant content inside a data set of this kind. A Hopfield model is used as a base for implementing this network, to which two weight matrixes are added: one constitutes the standard binary weights of the network, while the other one adds a bimodal weight component. Using both matrixes to process the input signals, the network achieves three possible states: -1, 1 and the brand-new 0-state, which represents a null, irrelevant or meaningless state. Equivalence between an input pattern and the network outputs during a recovery process using that same pattern (i.e. overlap) is taken as the main parameter to assess the performance of the three-state network. In a similar fashion to the weight matrix definition, overlap is divided between binary overlap *m* and bimodal overlap *l*. Tests performed inside the scope of this thesis show how both overlap values behave as the algorithm progresses learning new patterns. The introduction of the 0-state proves to be successful if parameter *l* is higher than *m* at the end of the test, as this indicates than the process of detecting relevant content before assigning it a sign value is more effective than simpler binary classification. Test outputs show a potential improvement on performance for one of the two data sets employed as input. However, due to the incompleteness of these data sets, further investigation should be performed in order to confirm these results.

# Palabras clave (castellano)

BGC, Bicuadrático, Bilineal, Bimodal, Binario, Corrupción, Hopfield, IPC, Modelado, Neurocomputación, Redes Neuronales, Red Neuronal Ternaria, Solapamiento, Topología, Transparencia Internacional

# Keywords (English)

Bilinear, Bimodal, Binary, Biquadratic, Corruption, CPI, GCB, Hopfield, Modeling, Neural Networks, Neurocomputing, Overlap, Ternary Neural Network, Topology, Transparency International

## *Agradecimientos*

A David Domínguez y a Hugo García, por guiarme y acompañarme en la realización de este trabajo. Sin su inestimable ayuda no habría sido capaz de completarlo.

A Judith Birkenfeld, por instigarme a no rendirme nunca ante la incertidumbre y ser una de esas tenues luces que iluminan el camino en la más vasta oscuridad.

A Carlos Jiménez y a Ramón García-Uceda por proporcionarme el apoyo emocional entre coetáneos que todos necesitamos.

Y por supuesto a mi madre, razón de mi ser y eterno bastión de apoyo y sustento a lo largo de todas las fases de mi vida.

# INDICE DE CONTENIDOS

# INDICE DE FIGURAS

# INDICE DE TABLAS

# 1 Introducción

## *1.1 Motivación*

La generalización de internet en los países desarrollados del mundo como nuevo medio de comunicación ha ocasionado que la información, y especialmente las noticias, se generen y se transmitan con una rapidez muy superior a cómo ocurría hace tan sólo dos décadas [4]; del mismo modo, el público ha logrado acceso a portales e informaciones que por su propia localización antes se encontraban inaccesibles, hecho que se ha visto amplificado por la normalización del inglés como lengua vehicular en occidente [1]. Como consecuencia, una gran parte de la ciudadanía de estos países ha visto homogeneizadas sus inquietudes al nutrirse de los mismos contenidos y ha comenzado a desarrollar una conciencia sobre los problemas y desafíos que conlleva vivir en una sociedad globalizada [3].

Uno de estos problemas es el de la corrupción institucional. Con el estallido de casos masivos como la trama Gürtel y sus procesos paralelos en España, o el de los papeles de Panamá en el ámbito internacional, ciertos organismos y figuras institucionales que tradicionalmente han formado parte fundamental del aparato estatal de muchos países han sufrido un grave revés en su credibilidad y la confianza que los ciudadanos depositaban en ellos se ha visto fuertemente deteriorada [5]. Transparencia Internacional, conocida ONG por sus denuncias contra la corrupción y el abuso de poder en las instituciones políticas, lleva desde el año 1995 elaborando y poniendo a disposición del público una serie de informes e indicadores sobre este asunto. En este trabajo se usan dos de ellos: el IPC (índice de percepción de la corrupción) [2], donde se recopila un agregado de las opiniones sobre la corrupción de los habitantes de más de 150 países; y el BGC (barómetro global de la corrupción) [1], donde a partir de investigaciones de elaboración propia, Transparencia realiza una cuantificación de la presencia de corrupción en las distintas instituciones de cada país.

Paralelamente, el desarrollo de nuevas tecnologías en los campos de la inteligencia artificial y la analítica, ha permitido a sociólogos, estadísticos y otros profesionales del sector trabajar mucho más rápido a la hora de evaluar e interpretar conjuntos de datos de grandes dimensiones [6]. En este trabajo se abordará el ejemplo de las redes neuronales artificiales (en concreto las redes de Hopfield) como modelo estructural con el que procesar datos de una manera y una eficiencia similar a como lo haría un cerebro humano. En la actualidad, las redes neuronales han adquirido protagonismo debido a que el aumento de la memoria y la capacidad computacional media de los ordenadores ha hecho viable su implementación [4][20]. Además se ha demostrado de forma práctica su efectividad para problemas de categorización y clasificación de datos y, en el caso de redes profundas o de gran número de capas ocultas, detección e identificación de rostros y objetos [7][8].

Este trabajo busca comprobar la efectividad de aplicar redes neuronales a la hora de detectar datos relevantes y realizar predicciones con indicadores sociológicos sobre corrupción como datos de entrada, cuantificar su rendimiento entre distintas implementaciones de la misma, y en última instancia, contribuir al posterior desarrollo de mejores sistemas para este uso.

## 1.2 Objetivos

El objetivo de este trabajo es el estudio del rendimiento de una red neuronal de Hopfield de tres estados (o ternaria) [9], en contraposición a una red binaria de las mismas características, empleando como datos de entrada indicadores sobre corrupción institucional. Para la realización de dicho estudio se variarán distintos aspectos de la red ternaria, modelándola así con el propósito de obtener una combinación de parámetros óptima sobre la cual realizar una comparativa del rendimiento de ambas redes a la hora de predecir datos de esta índole. La presente memoria se encargará de documentar todo el proceso y plasmar los resultados obtenidos, así como su interpretación.

## 1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- *Introducción*: breve descripción de los motivos para la realización de este trabajo, los objetivos y resultados esperados a la finalización del mismo, así como un sumario de la estructura de la presente memoria.
- *Estado de la cuestión*: trasfondo y detalles preliminares sobre la teoría y tecnologías usadas y evaluadas en el presente trabajo, bajo un enfoque contemporáneo y actualizado.
- *Diseño*: descripción exhaustiva de los procedimientos y activos empleados para el desarrollo de este proyecto, con las pertinentes justificaciones sobre las particularidades en la definición de los mismos.
- *Desarrollo*: documentación sobre la puesta en práctica y la implementación del diseño planteado en la sección anterior.
- *Pruebas y Resultados*: relación de las variables que definen las distintas pruebas a realizar, justificación y predicción teórica de los resultados sobre dichas pruebas y análisis empírico sobre los resultados obtenidos en la ejecución de cada una de ellas.
- *Conclusiones*: resumen de los resultados obtenidos, su interpretación sobre los objetivos del trabajo y posibles futuras líneas de trabajo a partir de los anteriormente citados.

# 2 Estado del arte

## *2.1 Estudio y prevención de la corrupción*

### 2.1.1 Antecedentes

La corrupción es un hecho cuya aparición se lleva documentando desde la antigüedad, y es inherente a prácticamente todas las culturas en las que existe una jerarquía de poder [13]. No obstante, la lucha activa contra la corrupción no se consolidó hasta bien entrados los años 90, con los comienzos del mundo global que hoy conocemos. La cristalización de unos valores comunes en la cultura occidental propiciaron y con ellos la aparición de organismos destinados a esta tarea; primeramente organizaciones privadas como es el caso de Transparencia Internacional, pero a partir de la presión ejercida por éstas, empezaron a crearse instituciones estatales en Europa y Latinoamérica que respondían al deseo cada vez más extendido de luchar contra la corrupción [3][12].

### 2.1.2 Transparencia Internacional

Transparencia Internacional es una ONG que se define a sí misma en su carta de valores como "un movimiento global que comparte la visión de un mundo en el que las instituciones de gobierno, comercio y sociedad civil estén libres de corrupción" [10]. Tiene sedes en más de 120 países, situándose su delegación en España en Madrid capital. Su historia se remonta a 1993 cuando Peter Eigen, ex director de banca de origen alemán, junto con nueve colaboradores de distintos países occidentales la fundó para concienciar sobre un tema que entonces era tabú: la corrupción y sus efectos en las distintas instituciones de la sociedad. Desde sus inicios, TI luchó en distintos países del mundo para dar a conocer su misión y convertir la lucha contra la corrupción en un asunto de calado a nivel internacional. Con el tiempo, la organización fue adquiriendo relevancia y reconocimiento, especialmente gracias a sus conexiones con organizaciones transnacionales como el Banco Mundial. La organización saltó a la fama en 1999 por denunciar casos de sobornos a estados por parte de empresas multinacionales para ganar contratos públicos. Con ello logró que los países miembros de la OCDE establecieran una convención para que éste tipo de actos pudieran castigarse legislativamente. Finalmente en el año 2003, la propia ONU estableció la UNCAC, primera convención en contra de la corrupción al más alto nivel institucional, la cual fue firmada por 140 países. En los años posteriores TI ha seguido fiel a su misión denunciando casos de corrupción en aquellos países donde se siguen produciendo estos hechos por falta de compromiso institucional o carencias propias del estado de dicha región. Y en los países donde sí se ha adoptado un compromiso, implicando a más actores y fomentando la participación de empresas, gobiernos y cuerpos internacionales para alcanzar el objetivo final de TI que es erradicar la corrupción por completo [11][12].

Además de participar activamente contra la corrupción en estrecha colaboración con organismos públicos y empresas, y organizando talleres de voluntariado y actividades diversas para concienciar y captar socios a nivel local, TI también contribuye a su misión realizando estudios e investigaciones sociológicas. A través de ellos, la organización publica de forma periódica una serie de informes e indicadores destinados a reflejar el estado actual en que se encuentra el mundo en materia de corrupción, abarcando diversos ámbitos, públicos y privados. Ejemplos de estas publicaciones a nivel internacional son el Índice de Fuentes de Soborno (IFS), el Informe Global de la Corrupción (IGC), o los dos

que se van a tratar en este trabajo: el Índice de Percepción de la Corrupción (IPC) y el BGC (Barómetro General de la Corrupción). En España, se publican adicionalmente índices sobre instituciones locales como el Índice de los Ayuntamientos (ITA), el Índice de las Comunidades Autónomas (INCAU), el Índice de las Diputaciones (INDIP) y otros sobre sectores muy concretos como el Índice de la Gestión del Agua (INTRAG) y el Índice de los Clubes de Futbol (INFUT) [14].

### 2.1.3 Índice de Percepción de la Corrupción

El índice de percepción de la corrupción (IPC) es un indicador que lleva siendo publicado de forma anual desde el año 1996 y es probablemente el trabajo más reconocible de TI. Se elabora a partir de los resultados de 13 estudios exhaustivos llevados a cabo los siguientes organismos internacionales: el Banco Africano de Desarrollo, la fundación Bertelsmann, el Banco Mundial, Economist Intelligence Unit, el Foro Económico Internacional, Freedom House, Global Insight, el IMD, PERC, el grupo PRS, VDEM, y World Justice Project [15]. Para la generación del indicador primero se estandarizan las fuentes de datos de forma relativa al tamaño de la muestra de cada país mediante la generación de variables normalizadas con datos de años anteriores, de manera que el indicador resultante sea un valor numérico oscilante entre 0 y 100. Este valor representa, de mayor a menor, el grado de precepción de la corrupción de ese país. Los países solamente se contabilizan en el IPC si han sido evaluados por al menos a 3 de los estudios propuestos. De ser el caso, el indicador final es la media resultante entre las variables incluidas en el estudio. Finalmente, se proporciona un intervalo de confianza máximo común al que pertenecen los datos generados para cada país [2][15].

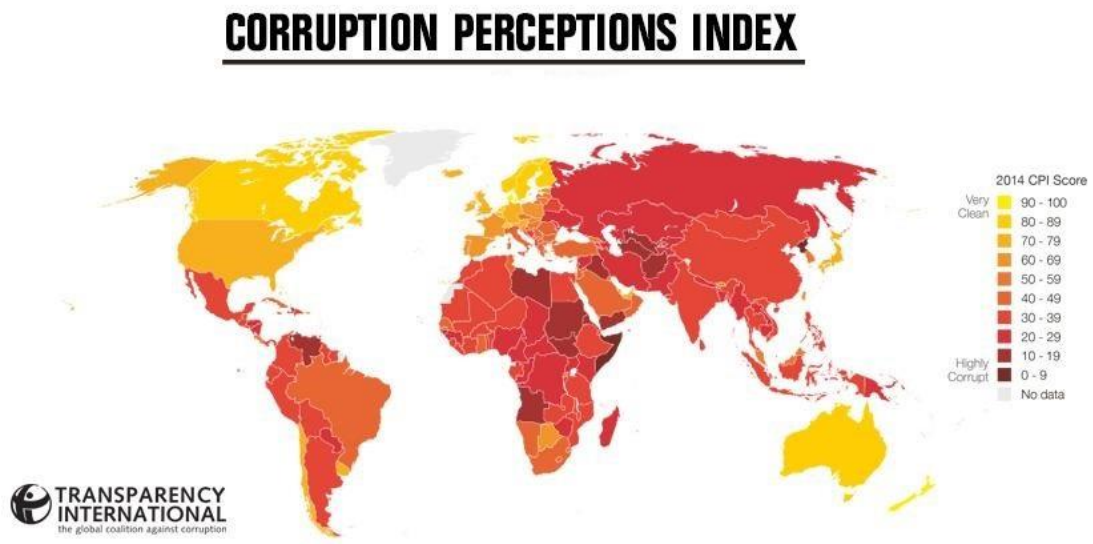



**Figura 2-1: Ejemplo de presentación de los datos del IPC para el año 2014**

Hasta el año 2012, el cálculo del IPC se normalizaba entre valores de 0 y 10, por lo que para el desarrollo de este trabajo, los datos anteriores a dicho año han tenido que ser escalados. No obstante, el cambio de escala perpetuado ha sido tan sencillo como multiplicar los valores antiguos por diez. Se puede apreciar coherencia en los datos escalados en comparación con años posteriores, puesto por su naturaleza secuencial en el tiempo no debería haber grandes variaciones, salvo eventualidades localizadas que supusieran un cambio sustancial en las instituciones del país implicado.

### 2.1.4 Barómetro General de la Corrupción

El barómetro general de la corrupción (BGC) es un indicador más exhaustivo que el IPC que se presenta con una periodicidad menor, normalmente por bienios. A diferencia del IPC, el BGC está basado en las opiniones de la población de más de 100 países en los que TI está presente y se elabora a partir de encuestas directas. En ellas, los participantes evalúan a partir de su propia visión la presencia de corrupción en 15 sectores de su país. Estos son: partidos políticos, sistema legislativo, sistema judicial, policía, empresas privadas, impuestos, aduanas, medios de comunicación, sistemas educativo y sanitario, servicios públicos y tramitación de permisos, fuerzas militares, ONGs y entidades religiosas. Los participantes evalúan cada uno de estos sectores con un valor numérico que varía entre 1 y 5, representando de mayor a menor el nivel de percepción percibido. El resultado final es un agregado de todas las opiniones recogidas [1][16].

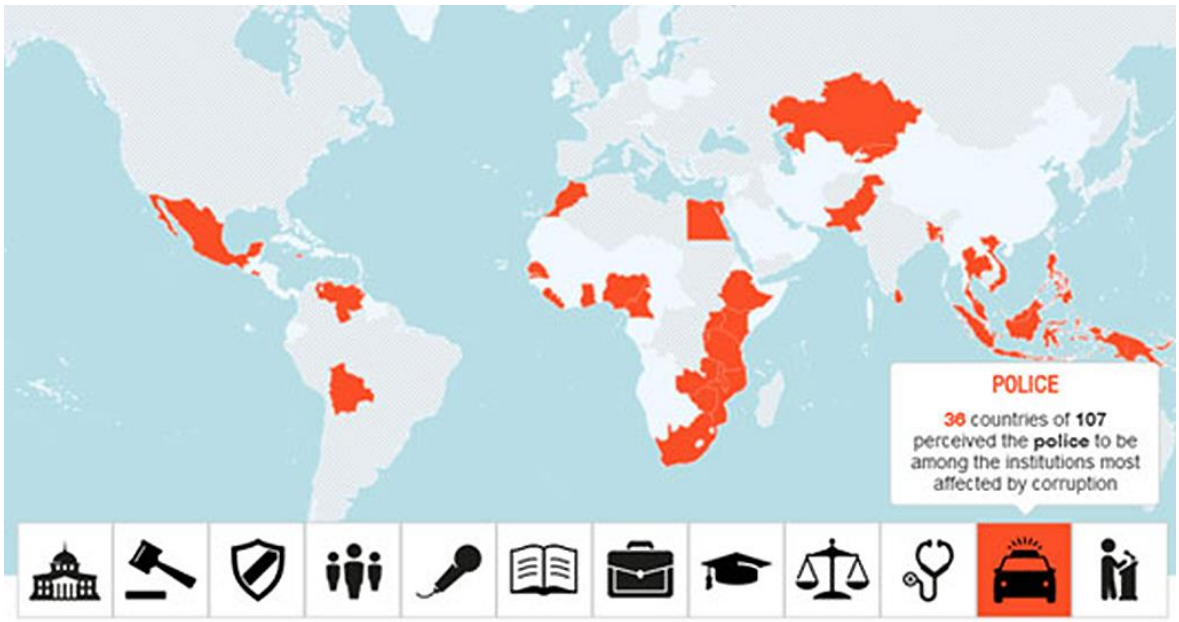



**Figura 2-2: Ejemplo de presentación de los datos del BGC sobre el sector policial para el año 2013**

Adicionalmente a los datos globales por país, TI presenta informes específicos por región, normalmente correspondientes a grupos de países radicados en un mismo continente o bloque socio-económico.

## *2.2 Neurocomputación*

### 2.2.1 Antecedentes

El término neurocomputación se refiere a la disciplina que estudia sistemas de procesamiento de la información no deterministas que se adaptan desarrollando asociaciones entre objetos, en respuesta a su entorno [17]. Estos sistemas también reciben el nombre de redes neuronales artificiales.

Las redes neuronales son asociaciones estructuradas de entidades matemáticas conocidas con el nombre de neuronas artificiales, las cuales fueron concebidas como un modelo que se comportara de manera análoga a las neuronas biológicas de las que se componen los cerebros de los organismos del reino animal y, por lo tanto, que explicase su funcionamiento. El concepto fue introducido al público en el año 1943 por Warren McCulloch y Walter Pitts en una publicación de la revista Bulletin of Mathematical Biophysics. En él se definió a la neurona artificial como una función matemática que recibe un sumatorio de entradas ponderadas, el cual luego es evaluado por una función no lineal para determinar el valor de salida [18]. Abstrayendo los conocimientos de la época sobre la biología del cerebro a partir de la doctrina desarrollada por Ramón y Cajal y otros investigadores entre finales del siglo XIX y principios del XX [19], las entradas del sistema representarían las dendritas del cerebro, receptores de las señales nerviosas de otras neuronas, mientras que la salida sería análoga a los axones.

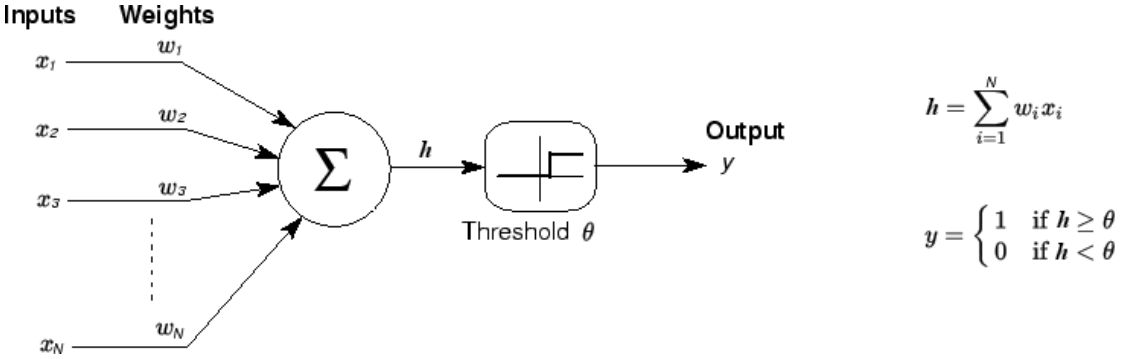



**Figura 2-3: Definición de una neurona artificial según el modelo de McCulloch-Pitts (MCP)**

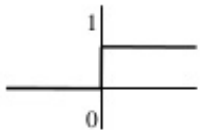
Este modelo inicial constituye la base que define los conceptos básicos de la neurocomputación y a partir del cual se han elaborado un gran número de investigaciones posteriores con el objetivo de extenderlo y/o complementarlo.

A finales de los años 60, un conocido estudio realizado por Marvin Minsky y Seymour Papert subrayó la impracticabilidad de las redes neuronales debido a que los computadores de la época no tenían suficiente memoria ni capacidad de procesamiento para trabajar con grandes volúmenes de datos [20]. Esto ocasionó que la investigación en este campo (y en general, todo lo referente a la Inteligencia Artificial) sufriera muchos recortes en financiación y por ello muchos investigadores la abandonaron, a pesar de que algunos continuaron y se han seguido haciendo progresos desde entonces. En la actualidad, las redes neuronales han recuperado protagonismo debido a que su implementación se ha convertido en algo plausible con la rápida evolución de las capacidades de los computadores modernos [4][17].

### 2.2.2 Comportamiento de una Neurona Artificial

Los modelos de redes neuronales derivados del modelo MCP están constituidos por neuronas que solamente generan dos estados de salida: activado y no activado. La determinación del estado de salida ***y*** de una neurona $\sigma_i$ depende en gran medida de cómo se defina la función de activación de la red. Una función de activación transforma el valor ponderado de las entradas de una neurona (o campo pre-sináptico $h_i$) en una salida válida en la red a través de una función matemática no lineal. Las más frecuentes son la función signo si los estados válidos son -1 y 1 o la función escalón si los estados válidos son 0 y 1.

$$step_t(x) = \begin{cases} 1 & x > \theta \\ 0 & otherwise \end{cases}$$




$$sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & otherwise \end{cases}$$

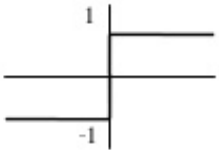



**Figura 2-4: Ejemplos de funciones de activación para redes neuronales binarias**

Existen una gran cantidad de funciones de activación aplicables como umbral de una red neuronal. Dependiendo de criterios como la naturaleza de los datos a manejar, si el ámbito de estos es discreto o continuo o si el objetivo de la red es clasificar datos o hacer predicciones, se deberá escoger con sumo cuidado la función más adecuada para lograr dicho objetivo, pues de ella depende como se comporte todo el sistema en conjunto [18] [31].

### 2.2.3 Aprendizaje en Redes Neuronales

Las redes neuronales artificiales están íntimamente relacionadas con el ámbito del aprendizaje automático. Con el ajuste de pesos en las diferentes conexiones de entrada de las neuronas de la red, se puede lograr que ésta modifique sus conexiones y en posteriores ejecuciones cambie su comportamiento y unifique sus salidas para entradas similares a las que ya han sido introducidas, siendo así capaz de identificar patrones de una misma naturaleza, como por ejemplo, rostros, voces u objetos similares [7][8].

Pocos años después de que apareciera el modelo MCP, Donald Hebb, psicólogo especializado en la correlación de las estructuras cerebrales con el comportamiento del individuo, propuso una teoría para explicar la como las neuronas se asociaban entre sí de las neuronas que se resume en el siguiente corolario [23]:

> "*Cuando el axón de una célula A se encuentra lo suficientemente cerca como para excitar repetidamente una célula B o toma parte en su activación de forma persistente, se produce cierto proceso de crecimiento o cambio metabólico en una o ambas células, de manera que la eficiencia de A aumenta cuando la célula activada es B*".
>
> Donald. O. Hebb, 1949

Este postulado fue aplicado de manera análoga al modelo MCP bajo la siguiente regla: si dos neuronas conectadas entre sí se activan de forma sincronizada, los pesos entre ellas deberán ser incrementados, lo cual puede expresarse con la siguiente formulación matemática [28]:

> *La variación de un peso de entrada **i** es el producto del valor de las entrada **i** de una neurona **x** por su valor post-sináptico **y** (o salida), multiplicado por una constante de aprendizaje **η**.*

$$\Delta w_i = \eta x_i y,$$

**Figura 2-5: Expresión generalizada de la regla de aprendizaje de Hebb**

La gran mayoría de modelos de redes neuronales, y especialmente aquellos que usan algoritmos de aprendizaje supervisado (i.e. aquellos capaces de predecir salidas para patrones con datos estructurados, debido a que los datos de entrada ***x*** producen salidas ***y*** expresables como una función de las entradas) [24], utiliza alguna variación de la regla de Hebb para calcular los pesos. Algunos ejemplos populares son: la regla delta, en la que las salidas de la red son ajustadas respecto a un modelo que las predice, utilizada por sistemas como el perceptrón lineal y ADALINE [25]; o el aprendizaje con olvido, utilizado en redes regresivas, en las que se aplica una constante con la que se consigue que la red deje de ser capaz de recuperar ciertos patrones aprendidos, normalmente los más antiguos, para que pueda seguir aprendiendo, a partir de la premisa de que una red neuronal tiene un número máximo de patrones que puede aprender antes de volverse ineficiente en sus predicciones, efecto conocido con nombre de sobreajuste [7].

A pesar de la relevancia de la teoría Hebbiana, el protagonismo adquirido en la actualidad por paradigmas neuronales innovadores (y ahora plausibles) como el Deep Learning, están desviando las investigaciones hacia el desarrollo de algoritmos de aprendizaje no Hebbianos [4][7][26][27].

### 2.2.4 Redes de Hopfield

En 1982, John Hopfield dio la definición de una red neuronal asociativa que buscaba servir como un modelo del comportamiento de la memoria humana. Contemporáneamente, este modelo se conoce con el nombre de *red de Hopfield* [21].

Una red de Hopfield puede definirse como un grafo compuesto de neuronas MCP, las cuales se encuentran conectadas entre sí con las siguientes restricciones [28]:

- Ninguna neurona puede estar conectada consigo misma
- El valor de los pesos de entrada para una conexión sináptica que va de una neurona ***i*** a otra neurona ***j*** es el mismo que el que va de una neurona ***j*** a otra neurona ***i***.

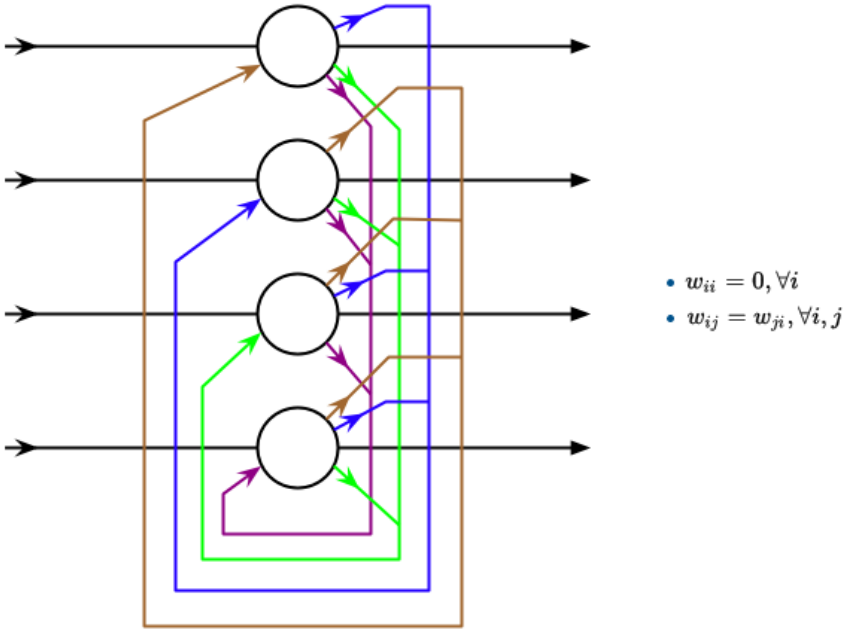



**Figura 2-6: Representación gráfica de una red de Hopfield**

Por la propia configuración de este tipo de redes, es típico representar los pesos de cada neurona como una matriz ***W*** de dimensión $n_{\times}n$, donde ***n*** es el número de neuronas de la red. Por consiguiente, cada uno de los elementos ***w*** que la componen representa el peso de una conexión dirigida desde la neurona $\sigma_i$ hasta la neurona $\sigma_j$. Siguiendo esta definición, la representación matricial de ***W*** en una red de Hopfield resulta en una matriz simétrica con una diagonal nula.

En casi todas sus implementaciones, el cálculo de los pesos $w_{ij}$ en una red de Hopfield se realiza mediante la aplicación de la regla de Hebb en su modalidad supervisada, aunque se ha demostrado que el uso del método de aprendizaje de Storkey permite que este tipo de redes adquieran una mayor carga de patrones máxima que con el aprendizaje Hebbiano [28][29].

$$w_{ij} = \frac{1}{n}\sum_{\mu=1}^{n} \epsilon_i^{\mu}\epsilon_j^{\mu} \qquad \text{donde } \epsilon_i^{\mu} \text{ representa el valor de la entrada } i \text{ para el patrón } \mu$$
$$\epsilon_i^{\mu} \{\pm 1\} \ o \ \epsilon_i^{\mu} \{0, 1\}$$

**Figura 2-7: Formulación de la regla aprendizaje de Hebb para una red de Hopfield**

La capacidad o carga máxima de una red de Hopfield es el número máximo de patrones que la red puede aprender antes de volverse ineficiente a la hora de recuperar patrones ya aprendidos o predecir salidas para nuevos patrones de entrada. En una red de Hopfield la carga $\alpha$ puede medirse según la fórmula siguiente:

$$\alpha = \frac{p}{k} \qquad \text{donde } p \leq k \leq n$$

**Figura 2-8: Expresión matemática de la carga de red en una red de Hopfield**

El parámetro $p$ corresponde al número de patrones aprendidos y $k$ es el número de conexiones efectivas de la red. El valor de $k$ suele corresponder al número total de neuronas de la red ($n$). Sin embargo, en el caso de una red de Hopfield diluida (i.e. en la que no todas las neuronas están conectadas entre sí), $k$ adquiere un valor menor que $n$ de manera que el ratio de carga $\alpha$ nunca sea mayor que 1, para que este no pierda su sentido teórico. Se ha demostrado empíricamente que el valor de carga máximo que una red de Hopfield admite antes de volverse ineficiente está próximo a $\alpha \approx 0.14$ [28][30].

### 2.2.5 Redes Neuronales de Tres Estados

Algunas de las investigaciones del campo de la neurocomputación se han dirigido al estudio y el desarrollo de modelos de red neuronal en los que las neuronas que los componen poseen un estado de salida adicional [33], sus pesos de entrada se restringen a tres valores [34] o incluso en los que se dan ambos casos simultáneamente [9]. Es lo que se conoce como *redes neuronales ternarias*.

El modelo que se va a estudiar en este trabajo [9] es un caso especial de la red de Hopfield en que tanto los atributos $\xi_i$ de cada patrón $\mu$, como las neuronas de la red $\sigma_i$, como así mismo los pesos de cada entrada están restringidos al ámbito de valores {-1, 0, 1}. La justificación para proponer un tercer estado en las neuronas es representar con él una situación de incertidumbre o de no relevancia de cara al procesamiento de datos con poca desviación entre sí, es decir, datos muy similares. Con esto se consigue diferenciar extraer aquellos que aportan contenido al conjunto de los que no, adquiriendo valores nulos éstos últimos, mientras que el resto serán representados en el rango {1, -1}

La red en cuestión se compone a su vez de dos matrices de pesos **W** y **V** de igual dimensión según lo especificado en la sección anterior. Esta duplicación respecto al modelo Hopfield original se efectúa para obtener el valor de salida de las neuronas $\sigma_i$ mediante un procedimiento compuesto:

- Con los elementos de **V** y la salida $y_i$ anterior de cada neurona se obtiene el campo pre-sináptico $g_i$, al que se le aplica la *función escalón* junto a un umbral cuadrático $u^c$, el cual suele ser función del conjunto de datos de entrada o de las propias neuronas. La salida indica si el valor de la neurona es nulo o no.
- Con los elementos de **W** y la salida $y_i$ anterior de cada neurona se obtiene el campo pre-sináptico $h_i$, al que se le aplica la *función signo* junto a un umbral lineal $u^l$, calculado a partir de los mismos parámetros que $u^c$. En caso de que la salida de la función de activación para $g_i$ no fuera nula, esta salida proporciona el signo del valor de la neurona.

Con la combinación de ambas operaciones sobre sus respectivos campos pre-sinápticos, se consigue el valor final para una neurona $\sigma_i$ en un contexto ternario.

$$\sigma_i^T = sign(h_i - u^l)\ step(|h_i| + g_i - u^c)$$

$$h_i = \sum_{j=1}^{N} w_{ij}\, y_j$$

$$g_i = \sum_{j=1}^{N} v_{ij}\, y_j^2$$

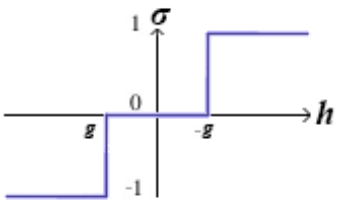



**Figura 2-9: Construcción de la salida de una neurona en una red de Hopfield de tres estados**

Para que esta definición de salida tenga sentido, el cálculo de cada elemento en las matrices de pesos ha de corresponderse con los valores esperados en cada componente (o modo) de la operación. Así pues, se realizan una serie de transformaciones previas a dicho cálculo:

- En el caso de la matriz de pesos binaria **W** el proceso de aprendizaje es *bilineal*. Puesto que los patrones de entrada $\boldsymbol{\mu}$ se componen de atributos $\xi_i^{\mu} \in \{-1, 0, 1\}$, hay que eliminar los estados nulos para que los valores que reciban las neuronas de la red se adecuen al ámbito binario $\sigma_i \{1, -1\}$. Para ello, dichos estados son sustituidos de manera aleatoria y uniforme por cualquiera de los otros dos.
- En el caso de la matriz de pesos bimodal **V** el proceso de aprendizaje es *bicuadrático*. En este caso se eleva al cuadrado cada atributo de los patrones de entrada $\boldsymbol{\mu}$ y se le resta el ratio de actividad *a* del patrón $\boldsymbol{\mu}$ de cada entrada $\xi_i^{\mu 2}$, esto es, el porcentaje de atributos que no tienen un valor nulo. Se obtienen así una serie de valores cuadráticos $\sigma_i^2 \in \{0, 1\}$. Con esta operación se elimina el estado "-1" por lo que el cálculo de pesos resultante es análogo al de una red de Hopfield binaria con función escalón como activación [32].

Una vez efectuadas las transformaciones pertinentes para obtener $\sigma_i$ y $\sigma_i^2$, se aplica la regla de Hebb de manera iterativa como un producto de todo $\sigma_i$ y $\sigma_i^2$ contra todo $\sigma_j$ y $\sigma_j^2$, respectivamente, y con dichos valores se actualizan los pesos $w_{ij}$ y $v_{ij}$ de ambas matrices, completando de este modo el proceso de aprendizaje de la red ternaria.

$$\Delta w_{ij}^{\mu} = \xi_i^{\mu} \xi_j^{\mu}$$

$$\Delta v_{ij}^{\mu} = (\xi_i^{\mu 2} - a)(\xi_j^{\mu 2} - a)$$

$$\mathbf{W}^{\mu} = \Delta \mathbf{W}^{\mu} + \mathbf{W}^{\mu - 1}$$

**Figura 2-10: Calculo de pesos para la red de Hopfield de tres estados**

# 3 Diseño

## 3.1 Arquitectura

Para realizar el estudio se ha diseñado un software capaz de medir el solapamiento de cada red entre el estado final sus neuronas y cada patrón de entrada para, de esta forma, comprobar efectivamente el rendimiento de cada una acorde a lo estipulado en el estudio base sobre el que se cimienta este trabajo [9]. Así pues, este software está configurado para recibir como entrada una serie de datos de las fuentes y características estipuladas en la sección 2.1, y devolver una evolución en el tiempo de los indicadores de solapamiento de la red, conforme ésta ha ido aprendiendo los patrones extraídos de dichos datos (véanse secciones 3.5 y 4.5).

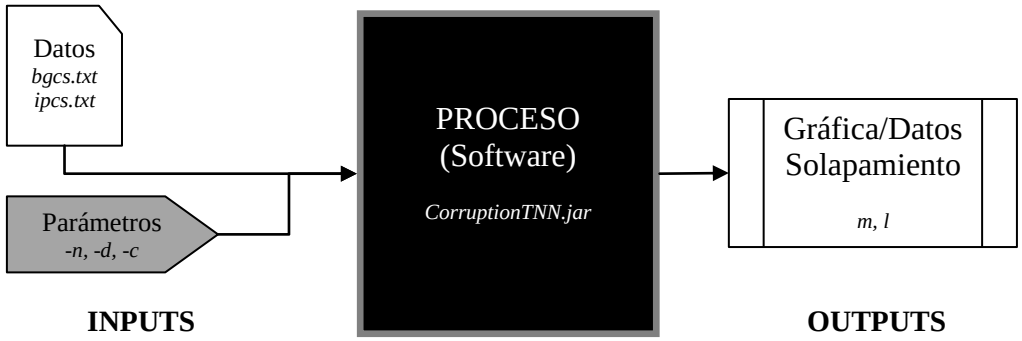



**Figura 3-1: Diagrama de caja negra del software planteado**

La arquitectura de este software se compone de cinco paquetes básicos:

- ***common***: constantes y parámetros globales, excepciones, tipos de datos propios del programa (enumeraciones) y métodos estáticos usados por los demás paquetes (útiles).
- ***input***: clases de almacenamiento y lectura de los datos de entrada
- ***neural***: implementaciones de las distintas redes de Hopfield (binaria, bimodal y ternaria)
- ***output***: clases de procesamiento y representación gráfica de los datos de salida
- ***test***: clases principales para la ejecución, determinan a través de la lectura de parámetros de entrada el comportamiento que debe tener el programa.

Las clases de *test* se encargan de crear las estructuras para almacenar los datos de entrada y salida y la red neuronal pertinentes para, a continuación, ejecutar el programa con variaciones en éstas estructuras y sus propios procedimientos, las cuales dependen de los parámetros de entrada que se hayan indicado. Por su parte, las clases de *neural* se encargan de realizar (con ayuda de los útiles) todas las operaciones vinculadas exclusivamente a la definición de la red, mientras que las clases de *input* y *output* se limitan a leer los ficheros de entrada cargándolos en una estructura de datos y procesar las salidas para su representación, respectivamente.
En las siguientes secciones se explica con detalle el objetivo y las funciones que cumplen las clases más relevantes de cada paquete. El diagrama de clases completo puede consultarse en el anexo B. Este programa ha sido construido enteramente en lenguaje *Java* usando el entorno de ejecución *JRE 1.7* [36].

## *3.2 Datos de Entrada*

### 3.2.1 Origen y Formato

Los datos de entrada son dos ficheros en formato texto que recogen respectivamente una relación de los indicadores del BGC y el IPC a través del tiempo. Dichos datos se pueden encontrar recogidos en las tablas sintéticas que se proporcionan en la web de TI [1][2].

En el caso de este trabajo, dichos datos fueron suministrados ya formateados como texto plano, a partir de un trabajo de fin de grado previo que también usó estos datos de entrada para realizar predicciones futuras de la corrupción [22]. Por ende, para esta operación sólo fue necesario desarrollar un módulo de lectura capaz de guardar los datos en estructuras de creación propia para ficheros con el formato especificado. No obstante, antes de su uso los ficheros de texto plano fueron normalizados manualmente con la eliminación de acentos, guiones y otros caracteres de los nombres de cada país. Del mismo modo, mientras que los datos del BGC siempre oscilaban entre valores de 1.0 y 5.0, los del IPC pasaron de variar entre 1.0 y 10.0 a 10 y 100 a partir del año 2012, por lo que hubo que ajustarlos para las pruebas, dividiendo los valores entre 10.
En la figura 3-2 se muestra un extracto de la tabla de datos original para el BGC y su correspondiente representación en texto plano, tras su normalización.

Sectores e Instituciones Nacionales * – ¿corruptos o transparentes?

| ¿En qué medida considera que los siguientes sectores se ven afectados por la corrupción en su país/territorio? (1: no corruptos, 5: muy corruptos) | Partidos Políticos | Parlamento/ Legislatura | Sistema Legal / Poder Judicial | Policía | Empresas/ sector privado | Impuestos | Aduanas | Medios | Servicios de Salud | Sistema Educativo | Registros y permisos | Servicios Públicos | Fuerzas Militares | ONGs | Entidades Religiosas |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Afganistán | 3,1 | 2,9 | 3,4 | 3,0 | 2,9 | 3,0 | 3,3 | 2,6 | 2,8 | 2,5 | 2,9 | 3,0 | 3,0 | 2,9 | 2,2 |
| Albania | 2,9 | 3,0 | 3,2 | 3,1 | 3,5 | 3,5 | 3,7 | 2,2 | 3,3 | 2,1 | 2,7 | 2,4 | 2,0 | 1,8 | 1,9 |
| Argentina | 4,6 | 4,6 | 4,3 | 4,4 | 3,7 | 3,6 | 4,2 | 3,5 | 3,3 | 3,1 | 3,8 | 3,7 | 3,4 | 2,9 | 3,0 |
| Austria | 3,3 | 2,8 | 2,6 | 2,8 | 2,9 | 2,7 | 2,6 | 2,8 | 2,4 | 2,3 | 2,5 | 2,4 | 2,5 | 2,4 | 2,5 |
| Bolivia | 4,5 | 4,3 | 4,0 | 4,2 | 3,2 | 3,6 | 4,2 | 2,8 | 3,0 | 3,0 | 3,0 | 3,0 | 3,6 | 2,7 | 2,2 |
| Bosnia y Herzegovina | 4,3 | 4,1 | 4,0 | 3,9 | 3,8 | 3,3 | 4,0 | 3,1 | 3,8 | 3,5 | 3,1 | 2,7 | 2,3 | 2,5 | 2,5 |
| Brasil | 4,5 | 4,3 | 4,2 | 4,4 | 3,8 | 4,2 | 3,9 | 3,6 | 3,9 | 3,9 | 3,6 | 3,8 | 3,4 | 3,0 | 3,0 |

```
2004
Afganistan;3.1;2.9;3.4;3;2.9;3;3.3;2.6;2.8;2.5;2.9;3;3;2.9;2.2
Albania;2.9;3;3.2;3.1;3.5;3.5;3.7;2.2;3.3;2.1;2.7;2.4;2;1.8;1.9
Argentina;4.6;4.6;4.3;4.4;3.7;3.6;4.2;3.5;3.3;3.1;3.8;3.7;3.4;2.9;3
Austria;3.3;2.8;2.6;2.8;2.9;2.7;2.6;2.8;2.4;2.3;2.5;2.4;2.5;2.4;2.5
Bolivia;4.5;4.3;4;4.2;3.2;3.6;4.2;2.8;3;3;3;3;3.6;2.7;2.2
BosniaHerzegovina;4.3;4.1;4;3.9;3.8;3.3;4;3.1;3.8;3.5;3.1;2.7;2.3;2.5;2.5
Brasil;4.5;4.3;4.2;4.4;3.8;4.2;3.9;3.6;3.9;3.9;3.6;3.8;3.4;3;3
```

**Figura 3-2: Comparativa de formato de los datos de entrada**

### 3.2.2 Estructuras de datos

Los datos de entrada se almacenan en objetos *Pattern*, los cuales representan patrones de entrada que utilizará la red neuronal. Un objeto *Pattern* se compone a su vez de objetos *Attribute*, que representan los atributos de los que se compone un patrón de datos en el contexto de una red neuronal. Puesto que en el caso de este estudio los atributos están multievaluados, es decir, los atributos se componen a su vez de varios valores, se definió que la clase *Attribute* se compusiera a su vez de un array auxiliar de valores, definidos como objetos *Double* por la naturaleza de los datos de entrada. De esta forma, se pueden almacenar este tipo de patrones con diversas jerarquías, pudiendo ser los elementos de más

alto nivel tanto años, como países, como sectores y quedando en última instancia almacenados los datos numéricos en los arrays auxiliares de cada atributo.

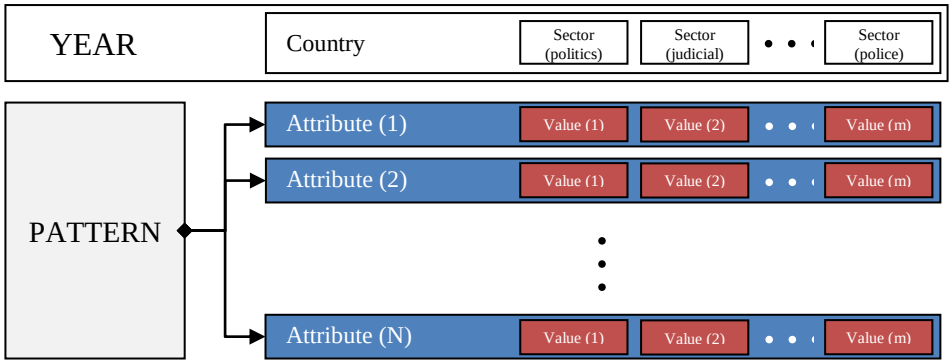



**Figura 3-3: Jerarquía de las estructuras de datos del paquete Input para el BGC**

### 3.2.3 Módulo de Lectura

La clase *InputDataReader* es la encargada de procesar los datos de entrada y guardarlos dentro de las estructuras previamente dadas. Independientemente del origen de los datos (BGC, IPC u otro), esta clase sigue el mismo procedimiento de carga en todos los casos:

1. Se mide el número de patrones ***p***, contando como referencia las líneas del fichero en las que sólo hay un elemento (en este caso, el año al que pertenecen los datos)
2. Se mide el número de atributos ***n*** máximo que un patrón puede tener:
    a. Guardando en un array auxiliar cada país único encontrado tras leer todas las líneas del fichero en todos los patrones
    b. Y evaluando el máximo número de sectores por país encontrados durante el mismo proceso de lectura. En el caso de los indicadores BGC e IPC se omite esta operación puesto que se sabe de antemano que se componen de 15 y 1 valores por atributo, respectivamente. Sin embargo, para otros tipos de datos es necesario guardar en una variable auxiliar este dato e ir actualizándola (si procediere) con cada iteración.
3. Se crea una estructura vacía *InputData*, análoga a un array de dimensión $\boldsymbol{p}_{\times}\boldsymbol{n}$, compuesta de ***p*** objetos *Pattern*, identificables por su año; y dentro de cada uno de ellos se crean ***n*** objetos *Attribute*, identificables por su país.
4. Se lee de nuevo el fichero y se ingresan los datos en la estructura *InputData*. Conforme se efectúa la lectura, cada vez que se llega a un nuevo patrón o atributo, se guarda el año o el país al que corresponden los datos que se están leyendo en ese momento en una variable auxiliar, que sirve para llamar a los objetos de los que se compone *InputData*. Los datos de sector se guardan en un array de objetos *Double* y tras llegar al fin de la línea, se copian dentro del objeto *Attribute* al que corresponden. Este proceso se repite iterativamente hasta que se llega al fin del fichero.

En algunos casos los datos de entrada proveen valores nulos, indicados con el carácter "*?*". Cuando esto ocurre, el programa les asigna el valor *0.0*, puesto que el rango de valores para ambos tipos de indicador en la práctica nunca llega a ese valor. Cuando los datos de entrada no son conocidos (i.e. ni BGC ni IPC), el valor asignado es *Double.NaN*. De cualquier manera, dada esta ocurrencia el programa convierte dichas entradas en valores aleatorios adecuados a la red neuronal con la que se haya decidido realizar el test.

## 3.3 Red Neuronal

### 3.3.1 Tipos de Red

Se han implementado dos modalidades de red neuronal de Hopfield, las cuales corresponden a las modalidades de red de Hopfield detalladas en la sección 2.2.5:

- ***BinaryHopfieldNetwork***: corresponde a una red de Hopfield de dos estados donde las neuronas $\sigma_i \in \{-1, 1\}$. Se emplea tanto para los test con red binaria pura y como componente binario de los test con red ternaria
- ***BimodalHopfieldNetwork***: corresponde a una red de Hopfield de dos estados donde las neuronas $\sigma_i^2 \in \{0, 1\}$. Se emplea como el componente bimodal de los test con red ternaria.

Ambas clases heredan de una clase básica *HopfieldNetwork* que contiene los parámetros y métodos necesarios para actuar acorde a la definición de red proporcionada en la sección 2.2.4 de este documento. La razón para establecer esta relación de herencia es que estructuralmente las dos redes funcionan de una manera muy similar; la única variación entre ellas está en las fórmulas de cálculo de las matrices de pesos y de sus respectivos campos pre-sinápticos. Por ello, se han definido *métodos abstractos* destinados únicamente a contener las fórmulas anteriormente citadas, para que puedan ser implementados individualmente por cada modalidad de red.

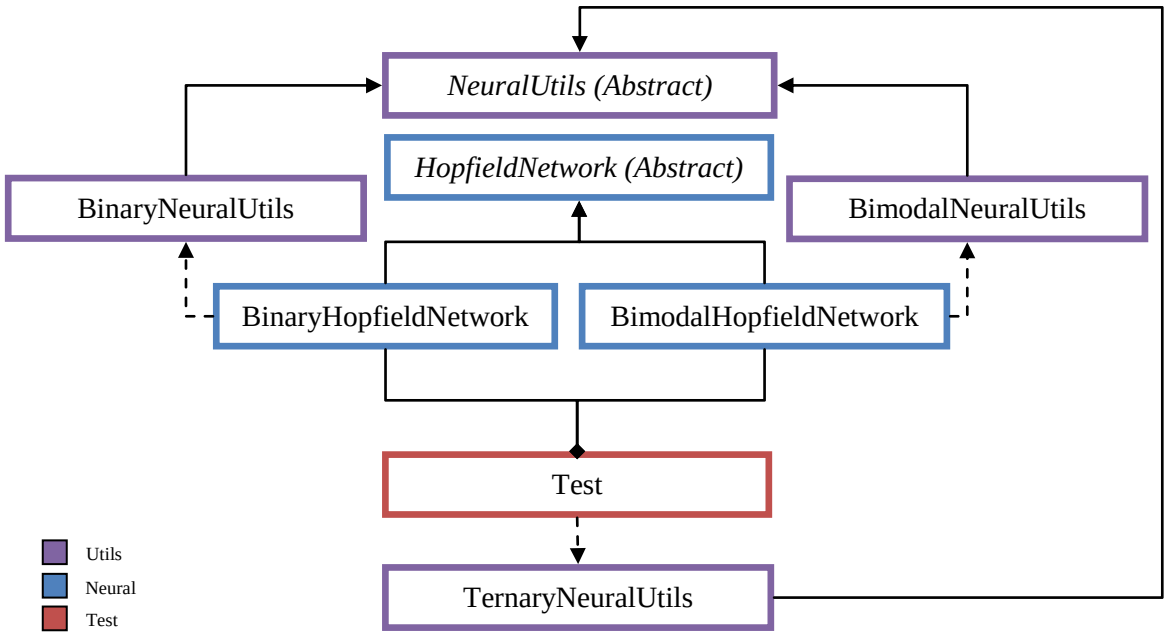



**Figura 3-4: Relaciones de la clase NeuralUtils con el resto de módulos del programa**

Adicionalmente, un objeto *HopfieldNetwork* contiene una instancia de la clase *NeuralUtils*, la cual no es más que un contenedor de métodos que actúan dentro del contexto de la red neuronal a la que están asociados pero no interactúan directamente sobre esta. La clase *NeuralUtils* sigue las mismas relaciones de herencia que *HopfieldNetwork*, dando lugar a las clases *BinaryNeuralUtils* y *BimodalNeuralUtils*, las cuales ejecutan los métodos adecuados al contexto de red neuronal al que están asociadas, como se puede apreciar en la figura 3-4. Estas clases están además íntimamente relacionadas con los módulos de test del programa, y especialmente la clase *TernaryNeuralUtils*, como se explicará en la sección 3.4.

### 3.3.2 Construcción de la Red

Cuando se llama al constructor de una de las redes detalladas en la sección anterior, se procesan y almacenan los siguientes parámetros:

- ***n*** – Número de neuronas de la red
- ***d*** – Fracción de *n* que se asigna a *k* (véase Figura 2-8). Representa el número de conexiones activas en una red diluida, i.e. cuando $k < n$.
- ***c*** – Constante de aprendizaje bicuadrática (véase sección 4.4)

A partir de estos parámetros se construyen los demás elementos que conforman la red:

- **Arrays de la red:** neuronas, campos pre-sinápticos y pesos. Usan *n* para fijar sus dimensiones y *c* para su cálculo posterior.
- **Semillas y parámetros topológicos**: se usa *k* para determinar que neuronas van a actuar y cuales no
- **Umbral *u***: una vez se obtienen los campos pre-sinápticos se fija bien como una función de estos o del estado de salida de las neuronas (véase sección 4.4 para una explicación más extensiva)

## *3.4 Módulos de Test*

### 3.4.1 Parámetros

Los módulos de test son la pieza central del programa. Dependiendo de parámetros como los citados en la sección anterior, indicaciones sobre el formato de entrada y salida de los datos, etc., la clase *TestFactory* se encarga de construir la instancia de la clase *Test* más adecuada. Dichos parámetros son fijados desde la propia entrada del programa. Estos son los argumentos que se pueden indicar al inicio de la ejecución:

- ***–input***: Tipo de fichero de entrada: *–bgc*/*–ipc* + Ruta de fichero de entrada
- ***–tn***: Indica que el tipo de test a ejecutar usa un contexto ternario o bimodal. Si no se indica se asume test en contexto binario.
- ***–extend***: Completar espectro de alfa con patrones de generación aleatoria (útil para generar gráficas con pocos patrones de entrada)
- ***–n***: Cantidad de ruido a utilizar (por defecto un 25%)
- ***–i***: Número máximo de iteraciones a realizar (por defecto 10) durante la fase de recuperación (véase sección 4.4)
- ***–d***: Porcentaje de neuronas a utilizar en la fase de recuperación
- ***–c***: Constante de aprendizaje bicuadrática para el cálculo de los campos pre-sinápticos (véanse secciones 3.4.2 y 4.4)
- **Flags de impresión**: según lo que se especifique, se muestran los datos de entrada recogidos, el estado de las neuronas a cada iteración, una tabla de resultados y/o la representación gráfica de ésta (más detalles en la sección 3.5)

Una relación completa de los argumentos que acepta el programa puede verse en el anexo A de este trabajo.

### 3.4.2 Tipos de Test

Existen tres variaciones de la clase Test usadas por este programa:

- ***BinaryTest***: usado en los test en los que los datos de entrada solamente son procesados por una red *BinaryHopfieldNetwork.* Dichos datos son previamente traducidos a formato ternario y pasados a formato binario (véase sección 4.2). Puesto que esta instancia de test solamente usa un contexto de red de Hopfield, no se tiene en cuenta el parámetro constante bicuadrática durante su ejecución. Su salida es el solapamiento binario (*m)* sobre el valor de carga de la red ($\alpha$).
- ***BimodalTest***: usado en los test en los que los datos de entrada solamente son procesados por una red *BimodalHopfieldNetwork* en un contexto ternario. Del mismo modo que con *BinaryTest*, los datos se traducen a ternario y se pasan a formato bimodal (véase sección 4.2). En este caso tampoco se considera el parámetro constante bicuadrática por las mismas razones que en el caso anterior. Su salida es el solapamiento bimodal (*l)* sobre el valor de carga de la red ($\alpha$).
- ***TernaryTest***: usado en los test que usan ambas instancias de *HopfieldNetwork* para calcular tanto el solapamiento binario como el bimodal. La constante de aprendizaje bicuadrática es considerada en este test únicamente si se indica. En caso contrario se usa su valor por defecto (véase sección 4.4).

**INSTANCIAS DE *TEST***

| | RED USADA | TRADUCCION DE DATOS | CONSTANTE BICUADRÁTICA (*c*) | SALIDA |
|---|---|---|---|---|
| ***BinaryTest*** | *BinaryHopfield Network* (sin parámetros) | Ternaria y conversión a $\sigma$ | c = 0.0 (obviada en la práctica) | Solapamiento binario (*m*) |
| ***BimodalTest*** | *BimodalHopfieldNetwork* (-t1) | Ternaria y conversión a $\sigma^2$ | c = 1.0 (obviada en la práctica) | Solapamiento bimodal (*l*) |
| ***TernaryTest*** | *Ambas* (-t2) | Ternaria y conversión $\sigma$ y $\sigma^2$ | c = 0.5 (por defecto) Ajustable por entrada | Solapamiento binario y bimodal (*m*/*l*) |

**Figura 3-5: Comparativa de las clases del paquete test**

## *3.5 Salida de Datos*

Este programa soporta distintos modos de salida, los cuales se detallan a continuación:

- **Estándar**: presenta una tabla sintética con los datos de solapamiento de la red en contraposición al valor de carga $\alpha$ actual tras haber aprendido *i* patrones. Salida por defecto.
- **Modo debug**: presenta de forma detallada todos los pasos y procesos efectuados por el programa, de manera análoga a la descrita en la sección 4.1. Se invoca con el parámetro *–pd*, y puede combinarse con los parámetros *–pm* (imprimir matrices de datos de entrada guardados como *InputData*) y *–pp* (imprimir patrones aprendidos y estados de las neuronas con cada iteración del programa)
- **Gráfico**: presenta una gráfica con el parámetro $\alpha$ como eje X y los valores de solapamiento como eje Y. Se invoca con el parámetro *–pg* y es combinable con cualquiera de los modos anteriores. Combinado con el parámetro –extend ajusta los rangos del eje X desde 0 hasta 1 y genera patrones aleatorios en caso de que no haya suficientes patrones de entrada como para completar el espectro de $\alpha$.
- **Gráfico suavizado**: análogo al modo anterior, pero los resultados que se presentan son promedios de una función media móvil sobre los datos de solapamiento obtenidos. Se invoca con el parámetro –psg y un dato que indica cuantos valores tomar como referencia a cada lado del punto actual para calcular la media.

# 4 Desarrollo

## 4.1 Procedimiento

Se presenta a continuación un diagrama de flujo del programa en el que se incluye por cada paso una descripción del mismo así como el paquete y clase encargados de ejecutarlo.

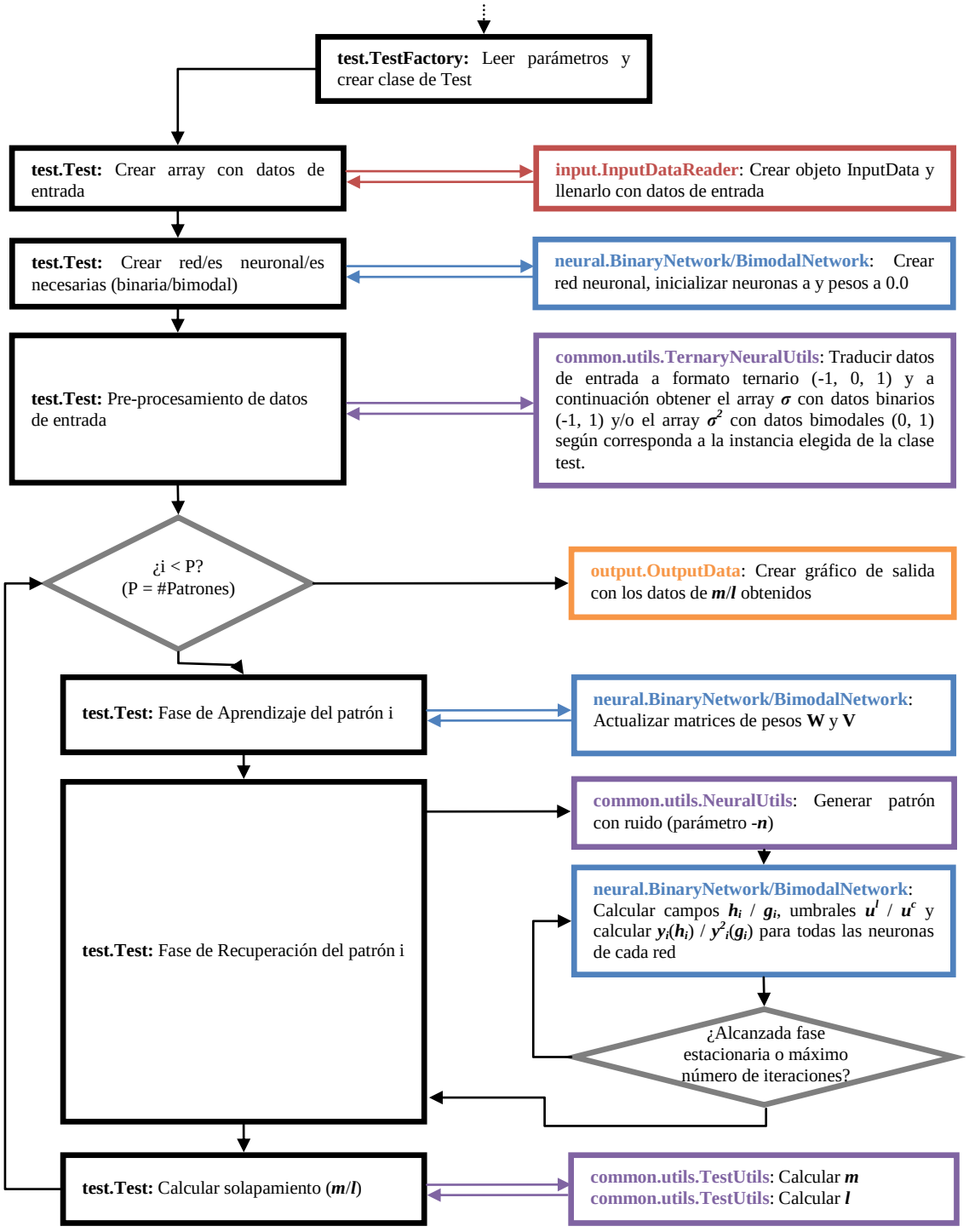



**Figura 4-1: Diagrama de flujo del programa**

## 4.2 Pre-procesamiento de los datos

Como se anticipaba en la sección 3.4.2 de este mismo documento, una vez leídos y almacenados los datos de entrada en un objeto *InputData*, estos son procesados y transformados antes de que la red o las redes neuronales asociadas al test en ejecución puedan trabajar con ellos. Este proceso es llevado a cabo de forma individual en cada patrón antes de entrar en el bucle principal, a través de tres métodos de la clase *TernaryNeuralUtils:*

- ***convertToNeuralFormat()***: se encarga de traducir los datos de entrada a un formato ternario para que cada atributo adquiera un valor $\xi_i \in \{-1, 0, 1\}$. Después, guarda la traducción del patrón en un nuevo array de objetos *Double*.
- ***generateSigma()***: a partir del array generado anteriormente, se genera la función $\sigma$ del patrón, con la que los atributos nulos pasan se transforman de manera uniforme en valores binarios de tal manera que cada atributo transformado tenga un valor $\sigma_i \in \{-1, 1\}$. Este array será manejado a continuación por la red *BinaryHopfieldNetwork*, si la configuración del test lo ha considerado, i.e. si el test no es una instancia de *BimodalTest*.
- ***generateSigma2()***: de igual manera que con la función anterior, se utiliza el array de atributos ternarios primeramente generado para obtener la función $\sigma^2$ del patrón, de forma que los atributos pasen a pertenecer a un ámbito bimodal tal que $\sigma^2_i \in \{0, 1\}$. El array resultante es manejado por la red *BimodalHopfieldNetwork* en todos los casos en que el test instanciado no sea puramente binario, es decir, de la clase *BinaryTest.*

## 4.3 Fase de aprendizaje

Cuando un patrón entra en esta primera fase del programa, la red o redes neuronales que se hayan inicializado en la construcción de la instancia de test pertinente, ejecutan el método *calculateWeights()*, con el cual se actualiza su matriz de pesos **W** o **V**. Este método recibe como parámetro de entrada la función $\sigma$ o $\sigma^2$ del patrón que haya sido generada anteriormente, como siempre dependiendo del tipo de red. A partir de esta entrada, la red actualiza cada elemento de dichas matrices utilizando el método de aprendizaje de Hebb descrito en la sección 2.2.4, cada atributo de $\sigma$ o $\sigma^2$ representa los atributos $\xi_i$ con los que se deriva la variación de los pesos para un patrón $\mu$. Terminada esta operación, se incrementa en 1 el parámetro interno *m* de la red, el cual representa el número de patrones aprendidos desde el comienzo del test (no confundir con el solapamiento binario de la red, también representado como *m*).

| | Parámetro de entrada | *calculateWeights()* |
|---|---|---|
| *BinaryHopfieldNetwork* | $\sigma$(Patrón $\mu$) | $w_{ij} = w_{ij} + \sigma^{\mu}_i \sigma^{\mu}_j$ |
| *BimodalHopfieldNetwork* | $\sigma^2$(Patrón $\mu$) | $v_{ij} = v_{ij} + \sigma^{\mu}_i{}^2 - a\ \sigma^{\mu}_j{}^2 - a$ |

**Tabla 4-2: Comparativa de operaciones realizadas durante la fase de aprendizaje del test**

## 4.4 Fase de recuperación

Esta es la fase principal del programa, en la que se efectúan las operaciones con la red neuronal que permitirán evaluar su rendimiento. Usando los mismos parámetros que los provistos y utilizados en la fase anterior, durante el proceso de recuperación se realizan las siguientes acciones:

- **Generación del patrón aprendido con ruido**: dependiendo de si la transformación del patrón que esté siendo procesada en base al test sea $\boldsymbol{\sigma}$, $\boldsymbol{\sigma^2}$ o ambas, se invoca a la implementación método *generateNoisyPattern()* a través de *BinaryNeuralUtils* o *BimodalNeuralUtils*, respectivamente. Este método modifica una fracción de los atributos de dichos arrays, determinada por el parámetro *–n* que haya sido introducido al inicio del test. Los valores que resultan modificados son sustituidos por su valor complementario dentro del contexto sobre el que estén definidos, esto es, -1 por 1 y viceversa para $\boldsymbol{\sigma}$, y 0 por 1 y viceversa para $\boldsymbol{\sigma^2}$.
- **Recuperación del patrón original a partir del generado con ruido en la fase anterior**. Se repiten las siguientes operaciones durante un número fijo de iteraciones dado por el parámetro *–i* a la entrada del programa:
    - **Cálculo de campos pre-sinápticos**: se obtiene el valor total de todas las señales de entrada de cada neurona según las definiciones provistas en la sección 2.2.5. Si la instancia del test es del tipo TernaryTest, se utiliza la constante bicuadrática provista como parámetro del programa *–c* durante el cálculo. Esta constante *c* sirve para ponderar el peso de cada modalidad de red a la hora de obtener los valores $\boldsymbol{h}$ y $\boldsymbol{g}$ y posteriormente recuperar el patrón ternario original. Es un factor crucial para determinar la configuración con la que una red de tres estados puede llegar a ser más eficiente que una red binaria común.
    - **Cálculo de Umbrales**: se calcula el umbral de la red o redes activas. En el caso de la red binaria se obtiene como la media de cada atributo transformado en $\boldsymbol{\sigma}$, mientras que en la red bimodal se obtiene a partir de los campos pre-sinápticos $\boldsymbol{g}$ sobre el ratio de actividad del patrón de entrada. Se puede añadir un parámetro $\boldsymbol{u^l}$ en la red binaria y $\boldsymbol{u^c}$ en la red bimodal para ajustar los umbrales y obtener así mejores resultados.
    - **Cálculo de valores de salida**: se obtiene el valor de salida $\boldsymbol{y}$ para cada neurona de cada red, aplicando el umbral $\boldsymbol{u^c}$ sobre cada campo pre-sináptico $\boldsymbol{h}$ o $\boldsymbol{g}$ tal y como se expresa en la figura 2-9, completando así un ciclo iterativo de esta fase.

  La fase concluye si se da alguno de estos dos supuestos: o bien los valores de salida coinciden exactamente con el patrón de entrada (se alcanza el *estado estacionario*), o bien se alcanza el máximo número de iteraciones *i* fijado al inicio.

| | *BinaryHopfieldNetwork* | | *BimodalHopfieldNetwork* | | Salida |
|---|---|---|---|---|---|
| | C. pre-sinápticos | Umbral | C. pre-sinápticos | Umbral | |
| *BinaryTest* | $\boldsymbol{h_i} = (1\text{-}c)\Sigma \boldsymbol{w_{ij}}\, \boldsymbol{\sigma_j}$ | $\boldsymbol{u^l} = \Sigma \boldsymbol{\sigma_i}/aN$ | - | - | $\boldsymbol{y_{bn}} = \boldsymbol{sign}(\boldsymbol{h_i} - \boldsymbol{u^l})$ |
| *BimodalTest* | - | - | $\boldsymbol{g_i} = (c)\Sigma \boldsymbol{v_{ij}}\, \boldsymbol{\sigma_i^2}$ | $\boldsymbol{u^c} = \Sigma \boldsymbol{g_i}/aN$ | $\boldsymbol{y_{bm}} = \boldsymbol{step}(\lvert\boldsymbol{h_i}\rvert + \boldsymbol{g_i} - \boldsymbol{u^c})$ |
| *TernaryTest* | $\boldsymbol{h_i} = (1\text{-}c)\Sigma \boldsymbol{w_{ij}}\, \boldsymbol{\sigma_j}$ | $\boldsymbol{u^l} = \Sigma \boldsymbol{\sigma_i}/aN$ | $\boldsymbol{g_i} = (c)\Sigma \boldsymbol{v_{ij}}\, \boldsymbol{\sigma_i^2}$ | $\boldsymbol{u^c} = \Sigma \boldsymbol{g_i}/aN$ | $\boldsymbol{y_t} = \boldsymbol{y_{bn}}\, \boldsymbol{y_{bm}}$ |

**Tabla 4-3: Comparativa de operaciones realizadas durante la fase de recuperación del test**

## 4.5 Cálculo del solapamiento

Finalmente, una vez termina la fase de recuperación se procede a calcular el solapamiento de la red. El solapamiento es una combinación lineal de las entradas del sistema con las salidas que representa como un ratio el porcentaje de atributos que la red o redes neuronales empleadas por el test han conseguido recuperar para un patrón $\mu$ del conjunto de datos de entrada. En el caso del solapamiento binario (*m*), esta evaluación se realiza sobre la totalidad del array de neuronas de dimensión *N*; pero en el caso bimodal (*l*), este parámetro se obtiene evaluando únicamente aquellas neuronas que posean actividad. En la figura 4-4 se presenta la definición de estos indicadores, dada en el estudio base sobre el que se cimienta este trabajo [9]:

$$m^{\mu}_{Nt} \equiv \frac{1}{aN}\sum_{i}\xi^{\mu}_{i}\sigma_{it}, \qquad l^{\mu} \equiv \frac{n^{\mu}-q}{1-a} =< \sigma^{2}\eta^{\mu} >,\ \eta^{\mu} \equiv \frac{(\xi^{\mu})^{2}-a}{a(1-a)},$$

**Figura 4-4: Definición matemática del solapamiento en una red neuronal para un patrón μ**

Según lo estipulado en la sección 2.2.4 de este documento, el grado de solapamiento que una red de Hopfield es capaz de obtener se va mermando conforme se van aprendiendo patrones de entrada, también expresado como conforme va incrementando la carga de red ($\alpha$). La salida de datos de este programa compara los valores de solapamiento binario y/o bimodal con cada posible valor de $\alpha$, según la configuración proporcionada por los datos de entrada, con el objetivo de reflejar si esta evolución corresponde a dicha premisa en el contexto de los datos empleados en este trabajo.

# 5 Integración, pruebas y resultados

## *5.1 Pruebas*

Tras considerar las especificaciones proporcionadas en los apartados de diseño y desarrollo de esta misma memoria, se pueden extraer tres parámetros que influyen de manera crítica en el rendimiento de los test propuestos, a partir de los cuales las pruebas han sido diseñadas. Estos son: el porcentaje de ruido que se le añade al patrón en la fase de recuperación, indicado a través del parámetro *–n* (sección 4.4); el número de conexiones activas de la red por cada neurona *k*, provisto como un parámetro *–d* que representa una fracción del número total de neuronas de la red (sección 3.3.2); y la constante de aprendizaje bicuadrática, para ponderar la intervención de la red binaria y la red bimodal a la hora de calcular la salida de una red de tres estados (sección 4.4), especificada con el parámetro *–c*. Así pues, las pruebas realizadas sobre los datos de entrada son una serie de combinaciones donde se evalúan diferentes combinaciones de valores para dichos parámetros.

Por la definición del programa, ***n*** indica el porcentaje de elementos del patrón que son invertidos por su valor complementario en la fase de pre-procesamiento, por lo que se espera que 0.0 y 1.0 tengan resultados idénticos pero con distinto signo, según la definición del funcionamiento de una red de Hopfield provista en las referencias de la sección 2.2.4. Por lo tanto, se espera que cuanto más cercano sea el valor de ruido a 0.5 peores sean los resultados obtenidos.

En las secciones de resultados solo se listan las gráficas obtenidas para ***c***=0.5, puesto que ***c***=0.0 y ***c***=1.0 son los únicos casos en los que la red ternaria muestra un comportamiento distinto, deteriorando severamente una de las dos modalidades de solapamiento pero dejando intacto el rendimiento de la otra; con cualquier otro valor la red se comporta de manera muy similar al primer caso citado, por lo que se ha optado por un parámetro 0.5 que para no mermar el rendimiento de ninguna de las dos redes y poder constatar los resultados de los test ternarios, que es el objetivo de este trabajo. En la interpretación de resultados, se toma *m* como el rendimiento obtenido por una red binaria pura, teniendo en cuenta que el valor máximo de solapamiento para ésta se alcanza cuando ***c*** es 0; y *l* como el rendimiento obtenido por la red bimodal. Por lo tanto, las gráficas se pueden interpretar por tanto como una comparativa entre red bimodal cuando ***c***=0.0, red ternaria cuando ***c***=0.5 y binaria cuando ***c***=1.0.

El número de iteraciones máximo que se ha fijado para las pruebas es de 50, puesto que en evaluaciones independientes se ha observado que en casi todos los casos la red alcanza la fase estacionaria antes de llegar a este número, y en los casos en los que lo sobrepasa los valores de solapamiento son muy bajos ($m, l < 0.3$).

Los resultados obtenidos para cada prueba están presentados en formato de gráfica suavizada (descrito en sección 3.5). Se ha utilizado un 20% del número total de patrones generados como factor de suavizado usando el modo completo (parámetro *–extend*, descrito en sección 3.4.1). Con esta configuración, los gráficos resultantes contemplan todo el espectro de alfa en cada test, obteniendo de esta forma unos resultados más exhaustivos.

## *5.2 Resultados*

### 5.2.1 IPC

Para el análisis de los datos del IPC se han empleado las siguientes variaciones de los tres parámetros principales:

- ***n*** = {0.0, 0.2, 0.35, 0.5}
- ***d*** = {1.0, 0.5, 0.25, 0.125, 0.08}
- ***c*** = {0.0, 0.5, 1.0}

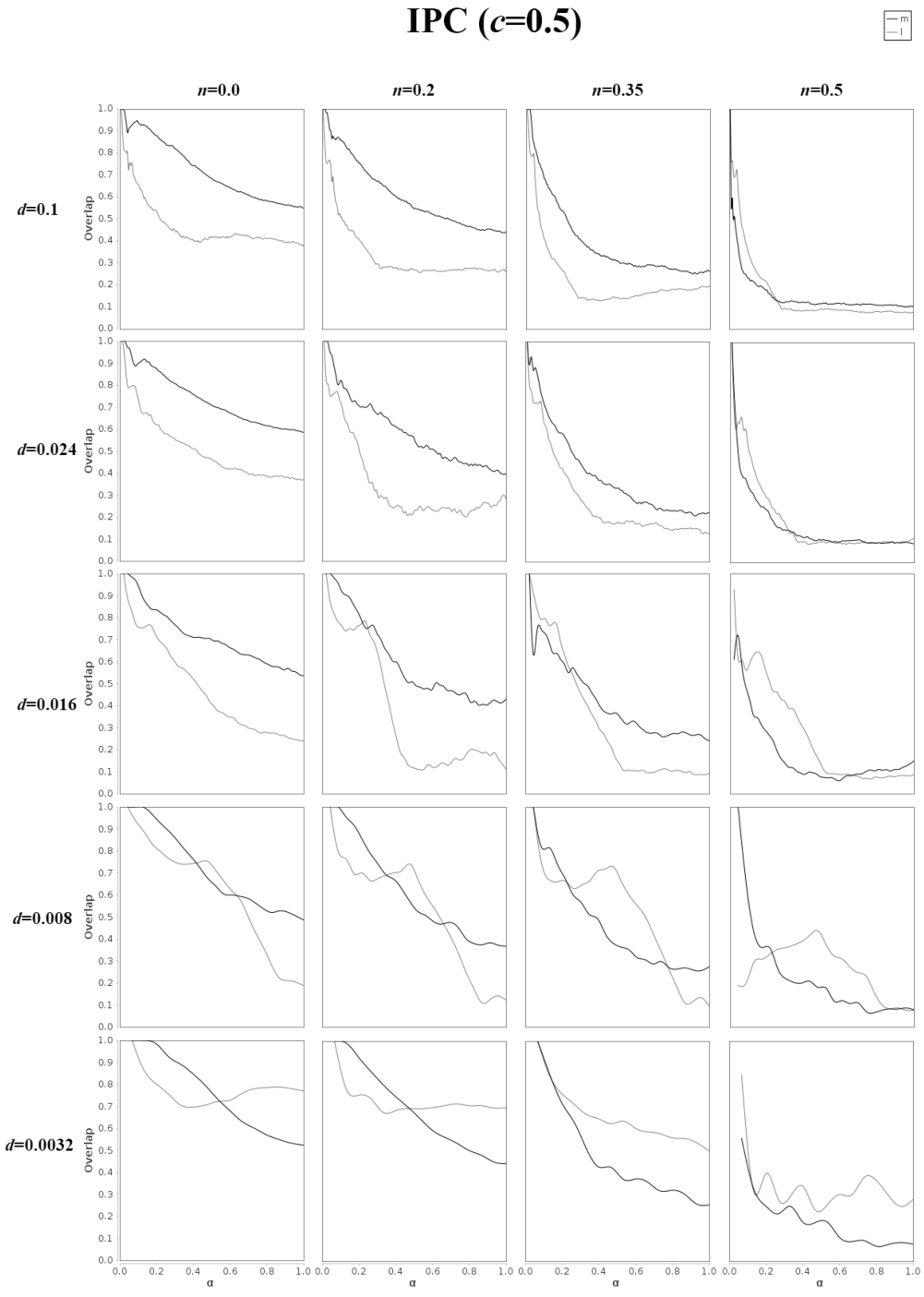



**Figura 5-1: Extracto de los resultados obtenidos con los datos del IPC como entrada**

Como observación más evidente, se puede ver en la evolución horizontal de las gráficas que conforme se va incrementando el nivel de ruido (***n***) decrece el rendimiento de la red tanto para el solapamiento binario *m* como para el solapamiento bimodal *l*, lo que parece lógico puesto que en el proceso de testado la red intenta recuperar un patrón aprendido. En el caso ***n***=0.5 la red neuronal se enfrenta a la máxima dificultad, tratando de recuperar un patrón que bien podría ser el aprendido o el invertido al aprendido. Conforme se van aprendiendo más patrones, este hecho se intensifica puesto que cada patrón dirime dos posibilidades, y el algoritmo se tarda cada vez más iteraciones en decidir que patrón recuperar, hecho que se puede ver en la evolución del solapamiento en las gráficas.

Con la variación de la topología de la red (donde ***d*** = *k*/*N*) ocurre algo interesante: conforme se disminuye el número de neuronas activas de la red aumenta el rendimiento de la red bimodal para mayores valores de carga de patrones $\alpha$. Esto no quiere intrínsecamente que con redes de menor tamaño se obtenga mayor eficiencia bimodal, y por ende ternaria. Es probable que este resultado se esté dando por que en el resto de casos se está completando el espectro de $\alpha$ con patrones aleatorios, ya que la muestra es tan solo *p*=190. La varianza interna en la distribución de atributos y las potenciales dependencias entre los datos de entrada del IPC y los patrones generados aleatoriamente (teniendo en cuenta que el programa garantiza que éstos sean uniformes) también pueden ocasionar que los valores de solapamiento, y en concreto el binario, se desvíe de la tendencia esperada conforme se incrementa la carga de la red.

### 5.2.2 BGC

Los parámetros usados en los test con los datos del BGC son los siguientes:

- ***n*** = {0.0, 0.2, 0.35, 0.5}
- ***d*** = {0.1, 0.024, 0.016, 0.008, 0.0032}
- ***c*** = {0.0, 0.5, 1.0}

En este caso se han elegido unos valores de ***d*** distintos puesto que la diferencia en la cantidad de atributos respecto al caso anterior hace que las fluctuaciones en la red sean mucho menos evidentes cuando se varía la topología. Como máximo se ha escogido 0.1, ya que a partir de dicho valor la red adquiere un comportamiento estable y no presenta fluctuaciones significantes en el solapamiento a mayor cantidad de neuronas activas. Esto se debe al elevado número de atributos que tienen los patrones de este conjunto de datos de entrada, lo que otorga robustez a la red; cabe decir además que el coste computacional para más casi 2000 patrones de igual número de atributos es muy elevado (tiempo cercano a 70 minutos por test), por lo que es más rentable trabajar con conjuntos más pequeños reduciendo la topología de la red.

En la siguiente página se puede observar que a diferencia de los resultados obtenidos con los datos del IPC, los datos para el BGC son muy consistentes entre sí, no sufriendo cambios perceptibles con la modificación de la topología, a pesar de llevarla a sus valores límite. Del mismo modo, el ruido ocasiona una disminución esperable del rendimiento de ambas redes, pero ningún valor extraño, consecuencia de la distribución propia de los datos de entrada. La causa más probable de esta eventualidad sea el reducido tamaño de la muestra de la que se ha dispuesto para realizar este estudio, al punto de que tan sólo se ha contado con seis patrones reales (p=6) y el resto han sido generados aleatoriamente. De ahí que los resultados con la red completa sean parecidos entre IPC y BGC y, por lo tanto, cupiera pensar que se ha producido el mismo efecto de desviación frente a la curva esperada que el ocurrido en el caso anterior.

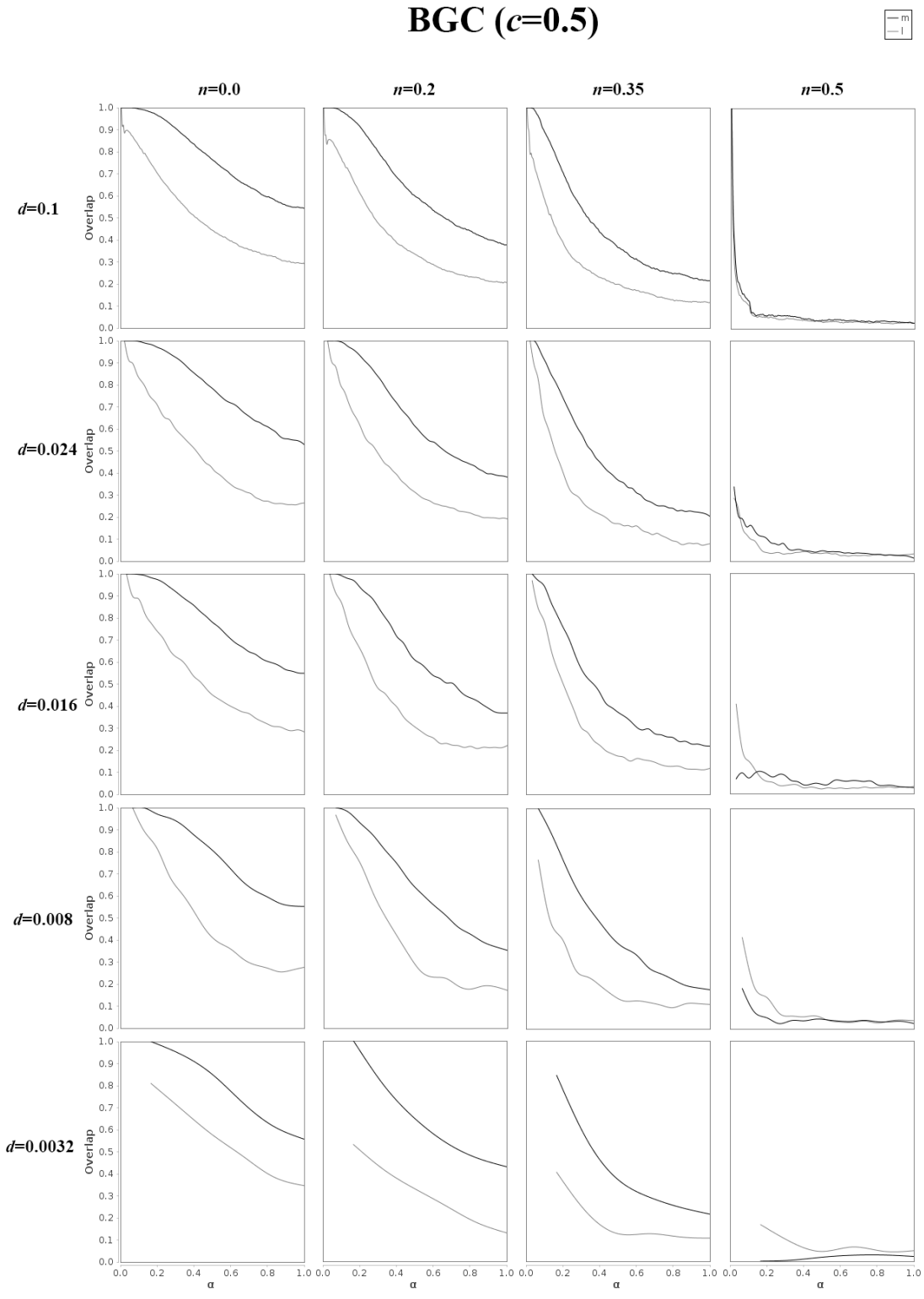



**Figura 5-2: Extracto de los resultados obtenidos con los datos del BGC como entrada**

### 5.2.3 Consideraciones adicionales

Como se venía adelantando en la sección 5.1, la variación del parámetro  no ha surtido grandes cambios en la presentación de resultados durante la realización de test independientes, por lo que no se ha considerado oportuno incluir las gráficas correspondientes a las distintas combinaciones de este parámetro con el nivel de ruido en el estímulo inicial ***n*** y la constante topológica ***d***. En la figura 5-3 se muestra una comparativa de los únicos casos en los que la variación de c surte algún efecto sobre el rendimiento de la red, neutralizando la componente binaria (en el caso ***c***=1.0) o bimodal (en el caso ***c***=0.0):

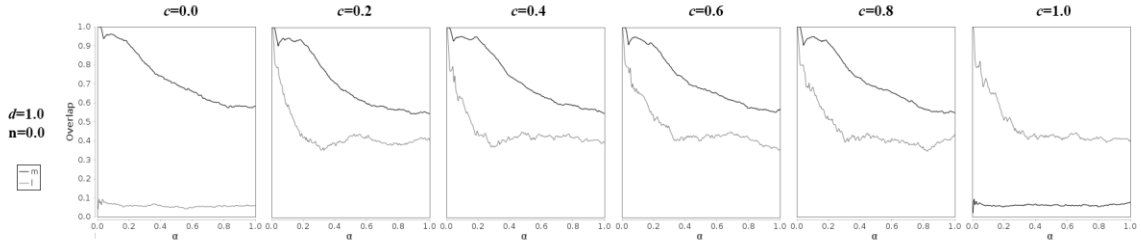



**Figura 5-3: Ejemplos de salida para la recuperación sin ruido de datos del IPC con distintos valores de *c***

Junto con los tres parámetros críticos considerados al inicio del apartado, otro factor de gran importancia en el comportamiento de la red es la función que se emplea para determinar el parámetro ***u***, como se detallaba en la sección 4.4 de este documento. En el caso de este estudio, se han probado brevemente el efecto de algunos valores de $\boldsymbol{u}^{\boldsymbol{c}}$ para el caso bimodal, por ser de interés mejorar su rendimiento para el objetivo final de este estudio. A continuación se detalla una breve relación de ellos:

- $\boldsymbol{u}^{\boldsymbol{c}}$ = 0: parámetro neutralizado. Empleado por defecto por los test presentados en las dos secciones anteriores al no haber encontrado ningún valor que mejore el rendimiento de la red bimodal.
- $\boldsymbol{u}^{\boldsymbol{c}}$ = s(***g***): como la desviación estándar de los campos pre-sinápticos bimodales. Su uso implicó una bajada más pronunciada del solapamiento bimodal *l* en todos los test independientes, especialmente en aquellos con una red muy diluida.

# 6 Conclusiones y trabajo futuro

## *6.1 Conclusiones*

En este trabajo se proponía investigar la efectividad de una red de Hopfield de tres estados frente a una red Hopfield binaria a la hora de evaluar y predecir datos sobre corrupción política, añadiendo a la misma una matriz de pesos bimodal con la que constituir una componente que diferenciara estados de contenido relevante frente a estados irrelevantes o nulos.

En todos los test se ha tratado de observar la tendencia de los parámetros de solapamiento *m* y *l* a lo largo de todo el espectro de $\alpha$, siendo necesario en prácticamente todos los casos la generación de patrones aleatorios uniformes, debido a las reducidas muestras de las que se ha dispuesto ($p_{IPC}$ = 15, $p_{BGC}$ = 6). Los resultados de las pruebas han indicado una efectividad superior de la red de tres estados cuando los datos de entrada pertenecen al IPC. Estos resultados se han hecho más patentes con el uso de redes muy diluidas donde el factor ***d*** tiende a hacer de *k* un valor muy cercano a *p*, es decir, en los casos en los que no se ha necesitado generar estos patrones aleatorios para ver el comportamiento de la red en todos los casos. Es por ello que existe la posibilidad de que los resultados finales se hayan visto afectados por la presencia de patrones aleatorios, cuyas distribuciones internas y dependencias entre sí sean muy dispares respecto de las de los indicadores políticos que estaban siendo evaluados.

Para el caso de los datos del BGC, se ha observado una gran cohesión en los resultados incluso cambiando el parámetro topológico ***d***. Ambos solapamientos de red solamente se han visto influidos por la inyección de ruido en los patrones a recuperar, y han seguido la tendencia de caída esperada para una red de Hopfield. Una de las razones principales para que se presente esta cohesión en los resultados es el alto número de atributos que posee cada patrón (*n*=1875), la cual viene avalada por los resultados de los test independientes con un factor topológico a ***d***=0.1, donde se han obtenido resultados casi idénticos a los obtenidos con ese valor ***d***. Sin embargo, no ha ocurrido como con el IPC, en ninguno de los casos (obviando aquellos sucedidos por un alto nivel de ruido) se ha visto que el solapamiento bimodal fuera superior al binario; aunque cabe remarcar que el conjunto de patrones de esta muestra *p* era aún más pequeño que en el caso del IPC por lo que el efecto descrito en el caso anterior puede haberse visto magnificado a mayor presencia de patrones aleatorios.

Adicionalmente, la variación del factor de aprendizaje ***c*** para ponderar el influjo de ambos cálculos (binario y bimodal) no ha causado grandes variaciones en los resultados, salvo en los casos extremos ***c***=0.0 y ***c***=1.0, en los que una de las dos modalidades ha sido neutralizada.

## *6.2 Trabajo futuro*

Los resultados obtenidos con los datos del IPC son prometedores y revelan que una red de tres estados podría ser más eficiente detectando contenido de esta índole cuando los indicadores han sido elaborados cohesivamente por instituciones especializadas, y no a partir de los datos por ciudadanos seleccionados de forma aleatoria como ocurre en el caso del BGC.

En contraposición, el reducido número de patrones con el que se ha contado para ambas pruebas ha hecho que los resultados sean también cuestionables. Por tanto, cabría realizar más investigaciones como la llevada a cabo en este estudio con otros indicadores de los que se cuente con más muestras, especialmente sobre aquellos con una gran cantidad de información, i.e. atributos. Otra opción sería combinar los datos del BGC y el IPC entre sí junto a otros indicadores de TI para obtener más datos, puesto que esta es una de las pocas organizaciones que provee datos especializados sobre corrupción institucional de forma abierta.

Respectivamente, en lo referente a la red de tres estados sería de interés realizar pruebas con distintos parámetros $\boldsymbol{u}$ para dirimir si es posible mejorar el rendimiento de alguna de las dos componentes, en especial la bimodal y, dentro de lo que es son los datos sobre corrupción, ver si alguna función de los elementos de la red se adecua mejor que otra a este tipo de contenidos.

# Referencias


[1] “Global Corruption Barometer”, Transparency International. Consultado el 18 de Mayo de 2017 http://www.transparency.org/research/gcb/gcb_2015_16

[2] “Corruption Perceptions Index”, Transparency International. Consultado el 18 de Mayo de 2017 http://www.transparency.org/research/cpi/overview

[3] L. Smith (ed.). “English for Cross-Cultural Communication”. MacMillan Press, 1981, pp. 211–216.

[4] M. Hilbert, P. Lopez. “The World’s Technological Capacity to Store, Communicate and Compute Information” in Science, volume 332 issue 6025, April 2011, pp. 60–65

[5] M. Villoria, G. G. Van Ryzin, C. F. Lavena. “Social and Political Consequences of Administrative Corruption: A Study of Public Perceptions in Spain” in Public Administration Review (ASPA), volume 73 issue 1, January 2013, pp. 85–94,

[6] H. Chen, R. H. L. Chiang, V. C. Story. “Business Intelligence and Analytics: from Big Data to Big Impact” in MIS quarterly, special issue: Business Intelligence Research, 2012

[7] A. Krizhevsky, I. Sutskever, G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks” in Advances in Neural Information Processing Systems 25 (NIPS 2012), December 2012

[8] C. Szegedy, A. Toshev, D. Erhan. “Deep Neural Networks for Object Detection” in Advances in Neural Information Processing Systems 26 (NIPS 2013), December 2013

[9] D. R. Domínguez Carreta, E. Korutcheva. “Three-state Neural Network: from Mutual Information to the Hamiltonian” in Physical Review E, volume 62 issue 2620, August 2010

[10] “Our Organization”, Transparency International. Consultado el 18 de Mayo de 2017 https://www.transparency.org/whoweare/organisation/

[11] “Our History”, Transparency International. Consultado el 18 de Mayo de 2017 https://www.transparency.org/whoweare/history

[12] P. Larmour. “Civilizing Techniques: Transparency International and the spread of anti-corruption”. Australian National University, Asia Pacific School of Economics and Government, paper 05–11, 2005

[13] M. I. Finley. “Politics in the Ancient World”. Cambridge University Press, 1983

[14] “Acerca de Transparencia Internacional España”, Transparencia Internacional. Consultado el 18 de Mayo de 2017 http://transparencia.org.es/acerca-de-ti-espana/

[15] “Corruption Perceptions Index 2016 – Methodology”, Transparency International. Consultado el 18 de Mayo de 2017 http://www.transparency.org/news/feature/corruption_perceptions_index_2016

[16] “Global Corruption Barometer 2013 – FAQ”, Transparency International. Consultado el 18 de Mayo de 2017 http://www.transparency.org/files/content/pressrelease/GCB2013_FAQs_EN.pdf

[17] R. Hecht-Nielsen. “Neurocomputing: picking the human brain” in IEEE Spectrum, volume 25 issue 3, March 1988, pp. 36–41

[18] W. McCulloch, W. Pitts. “A Logical Calculus of Ideas Immanent in Nervous Activity”. Bulletin of Mathematical Biophysics, volume 5 issue 4, December 1943, pp. 115–133

[19] S. Finger. "Chapter 13: Santiago Ramon y Cajal. From nerve nets to neuron doctrine" in Minds behind the brain: A history of the pioneers and their discoveries. New York: Oxford University Press. pp. 197–216.

[20] M. Minsky, S. Papert. “Perceptrons: An Introduction to Computational Geometry”, MIT Press, 1969
[21] J. Hopfield. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities” in Proceedings of the National Academy of Science of the USA, volume 79 issue 8, April 1982
[22] R. S. Fernández. “Predicción de la Corrupción vía Red Neuronal” (Trabajo de Fin de Grado), Febrero 2017, pp. 25–26
[23] D. O. Hebb. “The Organization of Behaviour: A Neuropsychological Theory”. John Wiley & Sons Inc. 1949.
[24] K. Kapitanova, S. H. Son. “Machine Learning Basics” in Intelligent Sensor Networks: The Integration of Sensor Networks, Signal Processing and Machine Learning, CRC Press, 2013, pp. 5–25
[25] B. Widrow, M. A. Lehr. "30 years of Adaptive Neural Networks: Perceptron, MADALINE, and Backpropagation" in Proceedings of the IEEE, volume 78 issue 9, 1990, pp. 1415–1442
[26] Y. LeCun, Y. Bengio, G. Hinton. “Deep Learning” in Nature, volume 521 issue 7553, May 2015, pp. 436–444
[27] Z. Shi, “Plasticity and Learning” Lecture 10 in Neuroinformatics. Institute of Computing Technology, Chinese Academy of Sciences, 2009
[28] “Hopfield Network”, Wikipedia.org. Consultado el 22 de Mayo de 2017 https://en.wikipedia.org/wiki/Hopfield_network
[29] A. Storkey. “Increasing the Capacity of a Hopfield Network without sacrificing Functionality” in Artificial Neural Networks (ICANN ’97), 1997, pp. 451–456
[30] J. Hertz, A. Krogh, R. G. Palmer. “Introduction to the Theory of Neural Computation”. Avalon Publishing, June 1991
[31] B. Karlik, A. V. Olgac. “Performance Analysis of Various Activation Functions in Generalized MLP Architectures for Neural Networks” in International Journal of Artificial Intelligence and Expert Systems, 2011.
[32] D. R. C. Dominguez. “The Antiquadrupolar Phase of the Biquadratic Neural Network” in Lecture Notes of Computer Science, volume 2686, June 2003
[33] H. Alemdar, V. Leroy, A. Prost-Boucle, F. Pétrot. “Ternary Neural Networks for Resource-Efficient AI Applications”. arXiv.org. Consultado el 23 de Mayo de 2017 https://arxiv.org/abs/1609.00222
[34] F. Li, B. Zhang, B. Liu. “Ternary Weight Networks”. arXiv.org. Consultado el 23 de Mayo de 2017 https://arxiv.org/abs/1605.04711
[35] S. Amari. “A Theory of Adaptive Pattern Classifiers” in IEEE Transactions on Electronic Computers, volume EC-16 issue 3, 1967, pp. 299–307
[36] “Java Platform Standard Edition 7 Documentation”. Oracle.com. Consultado en 24 de Mayo de 2017 http://docs.oracle.com/javase/7/docs/index.html

# Glosario

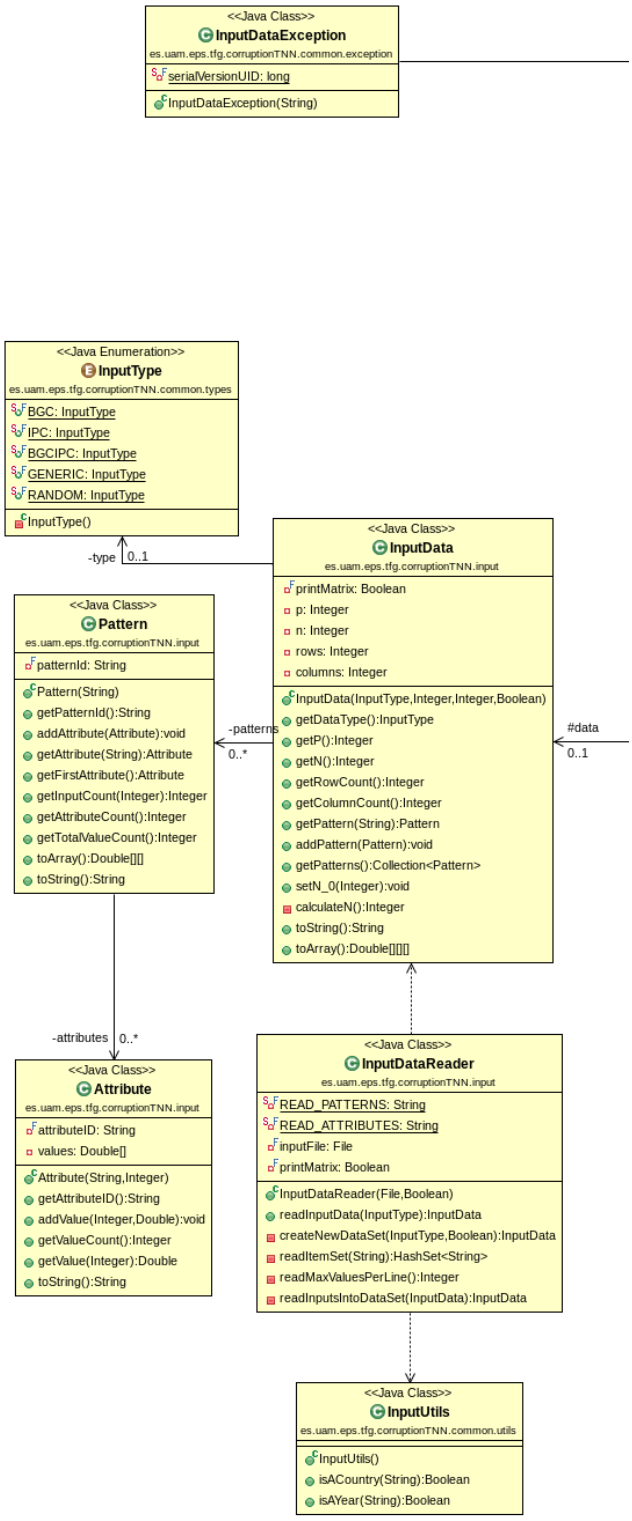
| | |
|---|---|
| BGC | Balance General de la Corrupción |
| CPI | Véase IPC |
| GCB | Véase BGC |
| IMD | International Institute for Management Development |
| IPC | Índice de Percepción de la Corrupción |
| MCP | McCulloch-Pitts (referido a neuronas o modelo de) |
| OCDE | Organización para la Cooperación y el Desarrollo Económicos |
| ONG | Organización No Gubernamental |
| ONU | Organización de las Naciones Unidas |
| PERC | Political & Economic Risk Consultancy LTD |
| PRS | Political Risk Services Group |
| TI | Transparencia Internacional |
| UNCAC | Convención de las Naciones Unidas contra la Corrupción (United Nations Convention against Corruption) |
| VDEM | Varieties of Democracy |

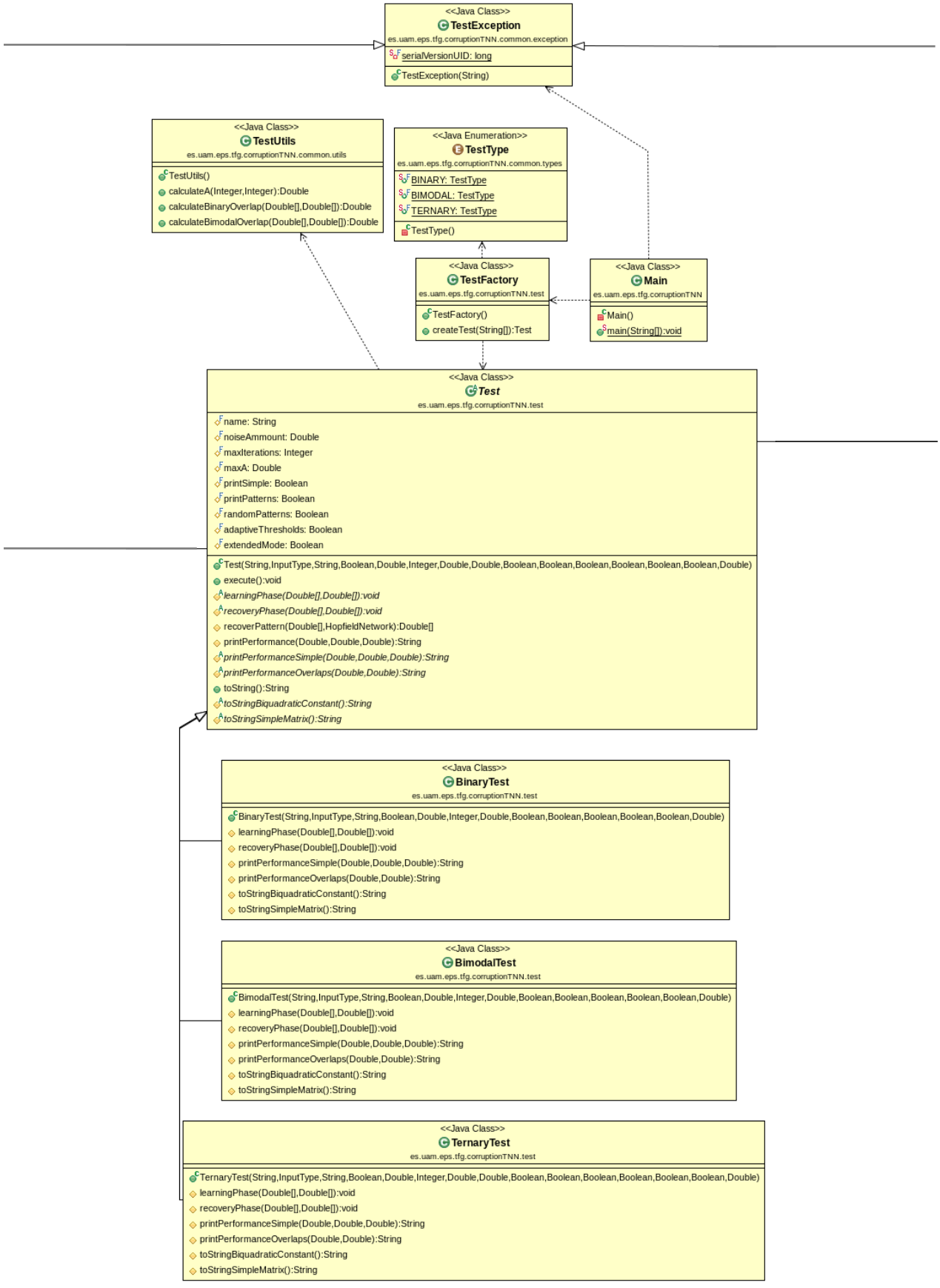
# Anexos

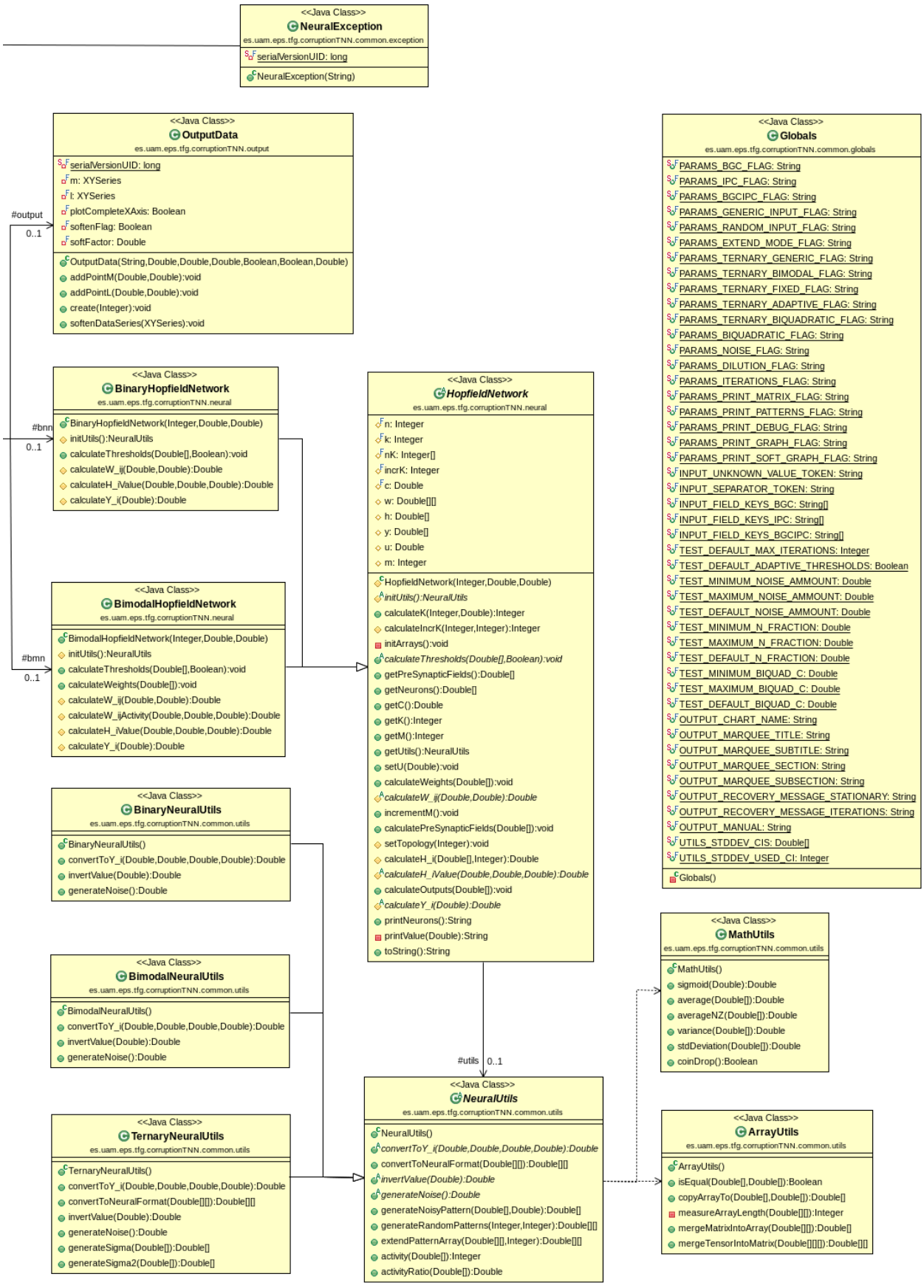
## A Manual de Uso

```
Intended use: java –jar CorruptionTNN.jar -module filePath [(-parameter [value])...]
       -module filepath: input data type to employ. Supported modes:
              -bgc: TI - Global Corruption Barometer data
              -ipc: TI - Corruption Perception Index data
              -input: generic/unknown input data (default)
       -extend [Optional]: execute test with extra generated patterns to obtain
        the complete alfa range. Best used to generate complete graphs with -pg
       -tn [Optional]: execute test using ternary neurons. If this parameter
        is not indicated binary neurons are used by default. Supported modes:
              -t1: only bimodal learning.
              -t2: bilineal and bimodal learning
       -c value [Optional with -t2]: biquadratic learning constant. Value
        can vary from 0 to 1. If not indicated the default value is used (0,5).
       -n value [Optional]: ammount of noise for recovery phase. Value can vary
        from 0.0 to 1.0. If not indicated, the default ammount is used (25%).
       -d value [Optional]: percentage of effective connections to use in the
        neural network (fraction of total attributes per pattern). Value can
        vary from 0.0 to 1.0. If ommitted, the default ammount is used (100%).
       -i [Optional]: sets the maximum number of iterations to take during
        the execution. If not specified, default value is used (10).
       -pd [Optional]: prints a detailed description of the test status during
        the whole process. Use this flag to debug the program and see how the
        input is processed and how the neurons change on every iteration.
       -pm [Optional]: display the data matrix before execution. Needs to be
        used with -pd to work.
       -pp [Optional]: show the patterns and the neurons status on each
        iteration. Needs to be used with -pd to work.
       -pg [Optional]: converts the results into a graph at the end of the
        test. Compatible with every other print option.
       -psg value [Optional]: analogue to -pg but the output graph is softened
        by a factor provided as value (acts as a moving average factor)
```

# B Diagrama de Clases




<<Java Class>>
TestException
es.uam.eps.tfg.corruptionTNN.common.exception
serialVersionUID: long
TestException(String)
<<Java Class>>
TestUtils
es.uam.eps.tfg.corruptionTNN.common.utils
TestUtils()
calculateA(Integer,Integer):Double
calculateBinaryOverlap(Double[],Double[]):Double
calculateBimodalOverlap(Double[],Double[]):Double
<<Java Enumeration>>
TestType
es.uam.eps.tfg.corruptionTNN.common.types
BINARY: TestType
BIMODAL: TestType
TERNARY: TestType
TestType()
<<Java Class>>
TestFactory
es.uam.eps.tfg.corruptionTNN.test
TestFactory()
createTest(String[]):Test
<<Java Class>>
Main
es.uam.eps.tfg.corruptionTNN
Main()
main(String[]):void
<<Java Class>>
Test
es.uam.eps.tfg.corruptionTNN.test
name: String
noiseAmmount: Double
maxIterations: Integer
maxA: Double
printSimple: Boolean
printPatterns: Boolean
randomPatterns: Boolean
adaptiveThresholds: Boolean
extendedMode: Boolean
Test(String,InputType,String,Boolean,Double,Integer,Double,Double,Boolean,Boolean,Boolean,Boolean,Boolean,Boolean,Double)
execute():void
learningPhase(Double[],Double[]):void
recoveryPhase(Double[],Double[]):void
recoverPattern(Double[],HopfieldNetwork):Double[]
printPerformance(Double,Double,Double):String
printPerformanceSimple(Double,Double,Double):String
printPerformanceOverlaps(Double,Double):String
toString():String
toStringBiquadraticConstant():String
toStringSimpleMatrix():String
<<Java Class>>
BinaryTest
es.uam.eps.tfg.corruptionTNN.test
BinaryTest(String,InputType,String,Boolean,Double,Integer,Double,Boolean,Boolean,Boolean,Boolean,Boolean,Double)
learningPhase(Double[],Double[]):void
recoveryPhase(Double[],Double[]):void
printPerformanceSimple(Double,Double,Double):String
printPerformanceOverlaps(Double,Double):String
toStringBiquadraticConstant():String
toStringSimpleMatrix():String
<<Java Class>>
BimodalTest
es.uam.eps.tfg.corruptionTNN.test
BimodalTest(String,InputType,String,Boolean,Double,Integer,Double,Boolean,Boolean,Boolean,Boolean,Boolean,Double)
learningPhase(Double[],Double[]):void
recoveryPhase(Double[],Double[]):void
printPerformanceSimple(Double,Double,Double):String
printPerformanceOverlaps(Double,Double):String
toStringBiquadraticConstant():String
toStringSimpleMatrix():String
<<Java Class>>
TernaryTest
es.uam.eps.tfg.corruptionTNN.test
TernaryTest(String,InputType,String,Boolean,Double,Integer,Double,Double,Boolean,Boolean,Boolean,Boolean,Boolean,Boolean,Double)
learningPhase(Double[],Double[]):void
recoveryPhase(Double[],Double[]):void
printPerformanceSimple(Double,Double,Double):String
printPerformanceOverlaps(Double,Double):String
toStringBiquadraticConstant():String
toStringSimpleMatrix():String


<<Java Class>>
NeuralException
es.uam.eps.tfg.corruptionTNN.common.exception
serialVersionUID: long
NeuralException(String)
<<Java Class>>
OutputData
es.uam.eps.tfg.corruptionTNN.output
serialVersionUID: long
m: XYSeries
l: XYSeries
plotCompleteXAxis: Boolean
softenFlag: Boolean
softFactor: Double
OutputData(String,Double,Double,Double,Boolean,Boolean,Double)
addPointM(Double,Double):void
addPointL(Double,Double):void
create(Integer):void
softenDataSeries(XYSeries):void
#output
0..1
<<Java Class>>
Globals
es.uam.eps.tfg.corruptionTNN.common.globals
PARAMS_BGC_FLAG: String
PARAMS_IPC_FLAG: String
PARAMS_BGCIPC_FLAG: String
PARAMS_GENERIC_INPUT_FLAG: String
PARAMS_RANDOM_INPUT_FLAG: String
PARAMS_EXTEND_MODE_FLAG: String
PARAMS_TERNARY_GENERIC_FLAG: String
PARAMS_TERNARY_BIMODAL_FLAG: String
PARAMS_TERNARY_FIXED_FLAG: String
PARAMS_TERNARY_ADAPTIVE_FLAG: String
PARAMS_TERNARY_BIQUADRATIC_FLAG: String
PARAMS_BIQUADRATIC_FLAG: String
PARAMS_NOISE_FLAG: String
PARAMS_DILUTION_FLAG: String
PARAMS_ITERATIONS_FLAG: String
PARAMS_PRINT_MATRIX_FLAG: String
PARAMS_PRINT_PATTERNS_FLAG: String
PARAMS_PRINT_DEBUG_FLAG: String
PARAMS_PRINT_GRAPH_FLAG: String
PARAMS_PRINT_SOFT_GRAPH_FLAG: String
INPUT_UNKNOWN_VALUE_TOKEN: String
INPUT_SEPARATOR_TOKEN: String
INPUT_FIELD_KEYS_BGC: String[]
INPUT_FIELD_KEYS_IPC: String[]
INPUT_FIELD_KEYS_BGCIPC: String[]
TEST_DEFAULT_MAX_ITERATIONS: Integer
TEST_DEFAULT_ADAPTIVE_THRESHOLDS: Boolean
TEST_MINIMUM_NOISE_AMMOUNT: Double
TEST_MAXIMUM_NOISE_AMMOUNT: Double
TEST_DEFAULT_NOISE_AMMOUNT: Double
TEST_MINIMUM_N_FRACTION: Double
TEST_MAXIMUM_N_FRACTION: Double
TEST_DEFAULT_N_FRACTION: Double
TEST_MINIMUM_BIQUAD_C: Double
TEST_MAXIMUM_BIQUAD_C: Double
TEST_DEFAULT_BIQUAD_C: Double
OUTPUT_CHART_NAME: String
OUTPUT_MARQUEE_TITLE: String
OUTPUT_MARQUEE_SUBTITLE: String
OUTPUT_MARQUEE_SECTION: String
OUTPUT_MARQUEE_SUBSECTION: String
OUTPUT_RECOVERY_MESSAGE_STATIONARY: String
OUTPUT_RECOVERY_MESSAGE_ITERATIONS: String
OUTPUT_MANUAL: String
UTILS_STDDEV_CIS: Double[]
UTILS_STDDEV_USED_CI: Integer
Globals()
<<Java Class>>
BinaryHopfieldNetwork
es.uam.eps.tfg.corruptionTNN.neural
BinaryHopfieldNetwork(Integer,Double,Double)
initUtils():NeuralUtils
calculateThresholds(Double[],Boolean):void
calculateW_ij(Double,Double):Double
calculateH_iValue(Double,Double,Double):Double
calculateY_i(Double):Double
#bnn
0..1
<<Java Class>>
HopfieldNetwork
es.uam.eps.tfg.corruptionTNN.neural
n: Integer
k: Integer
nK: Integer[]
incrK: Integer
c: Double
w: Double[][]
h: Double[]
y: Double[]
u: Double
m: Integer
HopfieldNetwork(Integer,Double,Double)
initUtils():NeuralUtils
calculateK(Integer,Double):Integer
calculateIncrK(Integer,Integer):Integer
initArrays():void
calculateThresholds(Double[],Boolean):void
getPreSynapticFields():Double[]
getNeurons():Double[]
getC():Double
getK():Integer
getM():Integer
getUtils():NeuralUtils
setU(Double):void
calculateWeights(Double[]):void
calculateW_ij(Double,Double):Double
incrementM():void
calculatePreSynapticFields(Double[]):void
setTopology(Integer):void
calculateH_i(Double[],Integer):Double
calculateH_iValue(Double,Double,Double):Double
calculateOutputs(Double[]):void
calculateY_i(Double):Double
printNeurons():String
printValue(Double):String
toString():String
<<Java Class>>
BimodalHopfieldNetwork
es.uam.eps.tfg.corruptionTNN.neural
BimodalHopfieldNetwork(Integer,Double,Double)
initUtils():NeuralUtils
calculateThresholds(Double[],Boolean):void
calculateWeights(Double[]):void
calculateW_ij(Double,Double):Double
calculateW_ijActivity(Double,Double,Double):Double
calculateH_iValue(Double,Double,Double):Double
calculateY_i(Double):Double
#bmn
0..1
<<Java Class>>
BinaryNeuralUtils
es.uam.eps.tfg.corruptionTNN.common.utils
BinaryNeuralUtils()
convertToY_i(Double,Double,Double,Double):Double
invertValue(Double):Double
generateNoise():Double
<<Java Class>>
MathUtils
es.uam.eps.tfg.corruptionTNN.common.utils
MathUtils()
sigmoid(Double):Double
average(Double[]):Double
averageNZ(Double[]):Double
variance(Double[]):Double
stdDeviation(Double[]):Double
coinDrop():Boolean
<<Java Class>>
BimodalNeuralUtils
es.uam.eps.tfg.corruptionTNN.common.utils
BimodalNeuralUtils()
convertToY_i(Double,Double,Double,Double):Double
invertValue(Double):Double
generateNoise():Double
#utils
0..1
<<Java Class>>
NeuralUtils
es.uam.eps.tfg.corruptionTNN.common.utils
NeuralUtils()
convertToY_i(Double,Double,Double,Double):Double
convertToNeuralFormat(Double[][]):Double[][]
invertValue(Double):Double
generateNoise():Double
generateNoisyPattern(Double[],Double):Double[]
generateRandomPatterns(Integer,Integer):Double[][]
extendPatternArray(Double[][],Integer):Double[][]
activity(Double[]):Integer
activityRatio(Double[]):Double
<<Java Class>>
ArrayUtils
es.uam.eps.tfg.corruptionTNN.common.utils
ArrayUtils()
isEqual(Double[],Double[]):Boolean
copyArrayTo(Double[],Double[]):Double[]
measureArrayLength(Double[][]):Integer
mergeMatrixIntoArray(Double[][]):Double[]
mergeTensorIntoMatrix(Double[][][]):Double[][]
<<Java Class>>
TernaryNeuralUtils
es.uam.eps.tfg.corruptionTNN.common.utils
TernaryNeuralUtils()
convertToY_i(Double,Double,Double,Double):Double
convertToNeuralFormat(Double[][]):Double[][]
invertValue(Double):Double
generateNoise():Double
generateSigma(Double[]):Double[]
generateSigma2(Double[]):Double[]

# C Documentación del código (Sumarios Javadoc)

es.uam.eps.tfg.corruptionTNN.common.utils

## Class ArrayUtils

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.common.utils.ArrayUtils

---

```
public class ArrayUtils
extends java.lang.Object
```

ArrayUtils class Contains methods for the processing of simple and multidimensional arrays.

**Since:**

1.3

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`NeuralUtils`, `MathUtils`

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `ArrayUtils()` |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `java.lang.Double[]` | `copyArrayTo(java.lang.Double[] dest, java.lang.Double[] orig)`<br>Copies the contents of one origin array into a destination one. |
| `java.lang.Boolean` | `isEqual(java.lang.Double[] a1, java.lang.Double[] a2)`<br>Compares the contents of two arrays and determines if they are equivalent. |
| `private java.lang.Integer` | `measureArrayLength(java.lang.Double[][] input)`<br>Measures the total number of items in a 2-dimensional array summing the lengths of each single dimension array contained on it. |
| `java.lang.Double[]` | `mergeMatrixIntoArray(java.lang.Double[][] input)`<br>Merges a 2-dimensional array into a single dimensional one. |
| `java.lang.Double[][]` | `mergeTensorIntoMatrix(java.lang.Double[][][] input)`<br>Merges a 3-dimensional array into a single dimensional one. |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.input

## Class Attribute

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.input.Attribute

---

```
class Attribute
extends java.lang.Object
```

Attribute class This object represents a row of values for an input file, which are stored into the `values` member. It also contains an ID that is assigned at the moment of the construction `attributeID`

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`Pattern`

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| `private java.lang.String` | `attributeID` |
| `private java.lang.Double[]` | `values` |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| `Attribute(java.lang.String attributeID, java.lang.Integer n)`<br>Class Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| `void` | `addValue(java.lang.Integer i, java.lang.Double v)`<br>Adds a value into the values array for a multi-evaluated attribute |
| `java.lang.String` | `getAttributeID()`<br>Returns the given ID for this Attribute object |
| `java.lang.Double` | `getValue(java.lang.Integer i)`<br>Returns the value given at a specific position in the value array for a multi-evaluated attribute |
| `java.lang.Integer` | `getValueCount()`<br>Returns the number of values a multi-evaluated attribute has |
| `java.lang.String` | `toString()`<br>toString |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.neural

## Class BimodalHopfieldNetwork

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork
    es.uam.eps.tfg.corruptionTNN.neural.BimodalHopfieldNetwork

---

```
public class BimodalHopfieldNetwork
extends HopfieldNetwork
```

BimodalHopfieldNetwork class Bimodal variation of the HopfieldNetwork class. Methods `initUtils()`, `calculateThresholds(Double[], Boolean)`, `calculateW_ij(Double, Double)`, `calculateH_iValue(Double, Double, Double)`, and `calculateY_i(Double)` are overridden by this class.

**Since:**

1.4

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`BimodalNeuralUtils`

### Field Summary

**Fields inherited from class es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork**

`c, h, incrK, k, m, n, nK, u, utils, w, y`

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `BimodalHopfieldNetwork(java.lang.Integer n, java.lang.Double d, java.lang.Double c)`<br>Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `protected java.lang.Double` | `calculateH_iValue(java.lang.Double w_ij, java.lang.Double x_j, java.lang.Double c)`<br>Calculates the pre-synaptic potential g_i for a neuron i of a BimodalHopfieldNetwork object |
| `void` | `calculateThresholds(java.lang.Double[] currentPattern, java.lang.Boolean adaptiveFlag)`<br>Calculates the Neuron thresholds using the pre-synaptic fields of the network `HopfieldNetwork.h` or the neurons status `HopfieldNetwork.y` depending on the kind of thresholds used in the current test (adaptive/fixed) |
| `protected java.lang.Double` | `calculateW_ij(java.lang.Double x_i, java.lang.Double x_j)`<br>Calculates a weight in a NeuralNetwork according to Hebb's law for supervised ANNs in a bimodal context |
| `protected java.lang.Double` | `calculateY_i(java.lang.Double g_i)`<br>Calculates the post-synaptic potential for a bimodal HopfieldNetwork |
| `protected NeuralUtils` | `initUtils()`<br>Initializes the associated Utils class of the network in a bimodal context |

**Methods inherited from class es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork**

`calculateH_i, calculateIncrK, calculateK, calculateOutputs, calculatePreSynapticFields, calculateWeights, getC, getK, getM, getNeurons, getPreSynapticFields, getUtils, incrementM, printNeurons, setTopology, setU, toString`

**Methods inherited from class java.lang.Object**

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.common.utils

## Class BimodalNeuralUtils

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils
    es.uam.eps.tfg.corruptionTNN.common.utils.BimodalNeuralUtils

---

```
public class BimodalNeuralUtils
extends NeuralUtils
```

BimodalNeuralUtils class Contains implementations for each abstract method of NeuralUtils using the context of a BimodalHopfieldNetwork

**Since:**

1.3

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

NeuralUtils

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| BimodalNeuralUtils() |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| java.lang.Double | convertToY_i(java.lang.Double x, java.lang.Double u, java.lang.Double sm, java.lang.Double sp)<br>Converts a value into a valid value for the neurons of a HopfieldNetwork |
| java.lang.Double | generateNoise()<br>Randomly returns a valid value for a bimodal context in order to generate noise in a Pattern. |
| java.lang.Double | invertValue(java.lang.Double ref)<br>Inverts the reference value using a bimodal context |

### Methods inherited from class es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils

activity, activityRatio, convertToNeuralFormat, extendPatternArray, generateNoisyPattern, generateRandomPatterns

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.test

## Class BimodalTest

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.test.Test
    es.uam.eps.tfg.corruptionTNN.test.BimodalTest

---

```
class BimodalTest
extends Test
```

### Field Summary

**Fields inherited from class es.uam.eps.tfg.corruptionTNN.test.Test**

adaptiveThresholds, bmn, bnn, data, extendedMode, maxA, maxIterations, name, noiseAmmount, output, printPatterns, printSimple, randomPatterns

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| BimodalTest(java.lang.String name, InputType inputType, java.lang.String filePath, java.lang.Boolean extendedMode, java.lang.Double noiseAmmount, java.lang.Integer iterations, java.lang.Double nFraction, java.lang.Boolean printSimple, java.lang.Boolean printMatrix, java.lang.Boolean printPatterns, java.lang.Boolean printGraph, java.lang.Boolean softenGraphFlag, java.lang.Double softenGraphFactor)<br>Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| protected void | learningPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the learning phase: 1. |
| protected java.lang.String | printPerformanceOverlaps(java.lang.Double m, java.lang.Double l)<br>Prints the overlap values obtained for the current iteration |
| protected java.lang.String | printPerformanceSimple(java.lang.Double a, java.lang.Double m, java.lang.Double l)<br>Prints the performance values for the current iteration in a format compatible with the simple results matrix |
| protected void | recoveryPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the recovery phase: 1. |
| protected java.lang.String | toStringBiquadraticConstant()<br>Prints the information about the biquadratic constant during this session. |
| protected java.lang.String | toStringSimpleMatrix()<br>Prints the headers for the simple results matrix depending on the type of Test instanced during this session. |

**Methods inherited from class es.uam.eps.tfg.corruptionTNN.test.Test**

execute, printPerformance, recoverPattern, toString

**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.neural

## Class BinaryHopfieldNetwork

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork
    es.uam.eps.tfg.corruptionTNN.neural.BinaryHopfieldNetwork

---

```
public class BinaryHopfieldNetwork
extends HopfieldNetwork
```

BinaryHopfieldNetwork class Binary variation of the HopfieldNetwork class. Methods `initUtils()`, `calculateThresholds(Double[], Boolean)`, `calculateW_ij(Double, Double)`, `calculateH_iValue(Double, Double, Double)`, and `calculateY_i(Double)` are overriden by this class.

**Since:**

1.4

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`BinaryNeuralUtils`

### Field Summary

**Fields inherited from class es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork**

`c, h, incrK, k, m, n, nK, u, utils, w, y`

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `BinaryHopfieldNetwork(java.lang.Integer n, java.lang.Double d, java.lang.Double c)`<br>Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `protected java.lang.Double` | `calculateH_iValue(java.lang.Double w_ij, java.lang.Double x_j, java.lang.Double c)`<br>Calculates the pre-synaptic potential h_i for a neuron i of a BinaryHopfieldNetwork object |
| `void` | `calculateThresholds(java.lang.Double[] currentPattern, java.lang.Boolean adaptiveFlag)`<br>Calculates the Neuron thresholds (set to 0.0 in a binary context) |
| `protected java.lang.Double` | `calculateW_ij(java.lang.Double x_i, java.lang.Double x_j)`<br>Calculates a weight in a NeuralNetwork according to Hebb's law for supervised ANNs in a binary context |
| `protected java.lang.Double` | `calculateY_i(java.lang.Double h_i)`<br>Calculates the post-synaptic potential for a binary network |
| `protected NeuralUtils` | `initUtils()`<br>Initializes the associated Utils class of the network in a binary context |

**Methods inherited from class es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork**

`calculateH_i, calculateIncrK, calculateK, calculateOutputs, calculatePreSynapticFields, calculateWeights, getC, getK, getM, getNeurons, getPreSynapticFields, getUtils, incrementM, printNeurons, setTopology, setU, toString`

**Methods inherited from class java.lang.Object**

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.common.utils

## Class BinaryNeuralUtils

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils
    es.uam.eps.tfg.corruptionTNN.common.utils.BinaryNeuralUtils

---

```
public class BinaryNeuralUtils
extends NeuralUtils
```

BinaryNeuralUtils class Contains implementations for each abstract method of NeuralUtils using the context of a BinaryHopfieldNetwork

**Since:**

1.3

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

NeuralUtils

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| BinaryNeuralUtils() |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| java.lang.Double | convertToY_i(java.lang.Double x, java.lang.Double u, java.lang.Double sm, java.lang.Double sp)<br>Converts a value into a valid value for the neurons of a HopfieldNetwork |
| java.lang.Double | generateNoise()<br>Randomly returns a valid value for a binary context in order to generate noise in a Pattern |
| java.lang.Double | invertValue(java.lang.Double ref)<br>Inverts the reference value using a binary context |

### Methods inherited from class es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils

activity, activityRatio, convertToNeuralFormat, extendPatternArray, generateNoisyPattern, generateRandomPatterns

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.test

## Class BinaryTest

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.test.Test
    es.uam.eps.tfg.corruptionTNN.test.BinaryTest

---

```
class BinaryTest
extends Test
```

### Field Summary

**Fields inherited from class es.uam.eps.tfg.corruptionTNN.test.Test**

adaptiveThresholds, bmn, bnn, data, extendedMode, maxA, maxIterations, name, noiseAmmount, output, printPatterns, printSimple, randomPatterns

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| BinaryTest(java.lang.String name, InputType inputType, java.lang.String filePath, java.lang.Boolean extendedMode, java.lang.Double noiseAmmount, java.lang.Integer iterations, java.lang.Double nFraction, java.lang.Boolean printSimple, java.lang.Boolean printMatrix, java.lang.Boolean printPatterns, java.lang.Boolean printGraph, java.lang.Boolean softenGraphFlag, java.lang.Double softenGraphFactor)<br>Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| protected void | learningPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the learning phase: 1. |
| protected java.lang.String | printPerformanceOverlaps(java.lang.Double m, java.lang.Double l)<br>Prints the overlap values obtained for the current iteration |
| protected java.lang.String | printPerformanceSimple(java.lang.Double a, java.lang.Double m, java.lang.Double l)<br>Prints the performance values for the current iteration in a format compatible with the simple results matrix |
| protected void | recoveryPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the recovery phase: 1. |
| protected java.lang.String | toStringBiquadraticConstant()<br>Prints the information about the biquadratic constant during this session. |
| protected java.lang.String | toStringSimpleMatrix()<br>Prints the headers for the simple results matrix depending on the type of Test instanced during this session. |

**Methods inherited from class es.uam.eps.tfg.corruptionTNN.test.Test**

execute, printPerformance, recoverPattern, toString

**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.common.globals

## Class Globals

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.common.globals.Globals

---

```
public class Globals
extends java.lang.Object
```

Globals class Contains all the program constants separated in categories: -Parameters: strings used by Main and TestFactory for parameter management -Input: strings used by InputDataReader to create an InputData object -Neural: constants used by HopfieldNetwork and NeuralUtils to correctly create the correct instance of a HopfieldNetwork (unused in this version) -Test: constants used by Test classes as default values during the execution of a test if no parameters are introduced when calling TestFactory. -Output: strings used by Test and OutputData to present test results and information about the execution of the program -Utils: constants used by the methods of the utils classes

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| static java.lang.String[] | INPUT_FIELD_KEYS_BGC |
| static java.lang.String[] | INPUT_FIELD_KEYS_BGCIPC |
| static java.lang.String[] | INPUT_FIELD_KEYS_IPC |
| static java.lang.String | INPUT_SEPARATOR_TOKEN |
| static java.lang.String | INPUT_UNKNOWN_VALUE_TOKEN |
| static java.lang.String | OUTPUT_CHART_NAME |
| static java.lang.String | OUTPUT_MANUAL |
| static java.lang.String | OUTPUT_MARQUEE_SECTION |
| static java.lang.String | OUTPUT_MARQUEE_SUBSECTION |
| static java.lang.String | OUTPUT_MARQUEE_SUBTITLE |
| static java.lang.String | OUTPUT_MARQUEE_TITLE |
| static java.lang.String | OUTPUT_RECOVERY_MESSAGE_ITERATIONS |
| static java.lang.String | OUTPUT_RECOVERY_MESSAGE_STATIONARY |
| static java.lang.String | PARAMS_BGC_FLAG |
| static java.lang.String | PARAMS_BGCIPC_FLAG |
| static java.lang.String | PARAMS_BIQUADRATIC_FLAG |
| static java.lang.String | PARAMS_DILUTION_FLAG |
| static java.lang.String | PARAMS_EXTEND_MODE_FLAG |
| static java.lang.String | PARAMS_GENERIC_INPUT_FLAG |
| static java.lang.String | PARAMS_IPC_FLAG |
| static java.lang.String | PARAMS_ITERATIONS_FLAG |
| static java.lang.String | PARAMS_NOISE_FLAG |
| static java.lang.String | PARAMS_PRINT_DEBUG_FLAG |
| static java.lang.String | PARAMS_PRINT_GRAPH_FLAG |
| static java.lang.String | PARAMS_PRINT_MATRIX_FLAG |
| static java.lang.String | PARAMS_PRINT_PATTERNS_FLAG |
| static java.lang.String | PARAMS_PRINT_SOFT_GRAPH_FLAG |
| static java.lang.String | PARAMS_RANDOM_INPUT_FLAG |
| static java.lang.String | PARAMS_TERNARY_ADAPTIVE_FLAG |
| static java.lang.String | PARAMS_TERNARY_BIMODAL_FLAG |
| static java.lang.String | PARAMS_TERNARY_BIQUADRATIC_FLAG |
| static java.lang.String | PARAMS_TERNARY_FIXED_FLAG |
| static java.lang.String | PARAMS_TERNARY_GENERIC_FLAG |
| static java.lang.Boolean | TEST_DEFAULT_ADAPTIVE_THRESHOLDS |
| static java.lang.Double | TEST_DEFAULT_BIQUAD_C |
| static java.lang.Integer | TEST_DEFAULT_MAX_ITERATIONS |

| | |
|---|---|
| static java.lang.Double | TEST_DEFAULT_N_FRACTION |
| static java.lang.Double | TEST_DEFAULT_NOISE_AMMOUNT |
| static java.lang.Double | TEST_MAXIMUM_BIQUAD_C |
| static java.lang.Double | TEST_MAXIMUM_N_FRACTION |
| static java.lang.Double | TEST_MAXIMUM_NOISE_AMMOUNT |
| static java.lang.Double | TEST_MINIMUM_BIQUAD_C |
| static java.lang.Double | TEST_MINIMUM_N_FRACTION |
| static java.lang.Double | TEST_MINIMUM_NOISE_AMMOUNT |
| static java.lang.Double[] | UTILS_STDDEV_CIS |
| static java.lang.Integer | UTILS_STDDEV_USED_CI |

## Constructor Summary

Constructors

| Modifier | Constructor and Description |
|---|---|
| private | Globals() |

es.uam.eps.tfg.corruptionTNN.neural

## Class HopfieldNetwork

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.neural.HopfieldNetwork

**Direct Known Subclasses:**

BimodalHopfieldNetwork, BinaryHopfieldNetwork

---

```
public abstract class HopfieldNetwork
extends java.lang.Object
```

HopfieldNetwork class This object represents a Neural Network according to Hopfield's model. It is composed of a single layer/array of neurons (y), a matrix of weights (w), an array for containing the presynaptic fields (h), and some topology parameters: n Total number of neurons k Total number of active connections nK Array of topological seeds (which number of neuron to begin with when processing the presynaptic field of a neuron in setTopology(Integer), inside calculatePreSynapticFields(Double[]) incrK Topological increment (calculate presynaptic value using only each incrK neurons starting from the corresponding element of nK) The network has also some runtime parameters like the thresholds for the neuron activation (u, the number of patterns learned in a session m, a learning constant for biquadratic learning c and the corresponding utils class utils

**Since:**

1.4

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

NeuralUtils

## Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| protected java.lang.Double | c |
| protected java.lang.Double[] | h |
| protected java.lang.Integer | incrK |
| protected java.lang.Integer | k |
| protected java.lang.Integer | m |
| protected java.lang.Integer | n |
| protected java.lang.Integer[] | nK |
| protected java.lang.Double | u |
| protected NeuralUtils | utils |
| protected java.lang.Double[][] | w |
| protected java.lang.Double[] | y |

## Constructor Summary

Constructors

| Modifier | Constructor and Description |
|---|---|
| protected | HopfieldNetwork(java.lang.Integer n, java.lang.Double d, java.lang.Double c)<br>Constructor |

## Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| protected java.lang.Double | calculateH_i(java.lang.Double[] x, java.lang.Integer i)<br>Calculates the pre-synaptic field of a Neuron layer by appliying Hebb's rule of Supervised Learning (h_i = wi*xi). |
| protected abstract java.lang.Double | calculateH_iValue(java.lang.Double w_ij, java.lang.Double x_j, java.lang.Double c)<br>Calculates the pre-synaptic potential for a neuron of a HopfieldNetwork |
| protected java.lang.Integer | calculateIncrK(java.lang.Integer n, java.lang.Integer k)<br>Calculates the value of incrK (number of neurons to skip per iteration when k < n) by calculating the floor of n/k |
| java.lang.Integer | calculateK(java.lang.Integer n, java.lang.Double d)<br>Calculates the value of k (number of connections to use) by using a delta parameter to get a fraction of N in the first layer |
| void | calculateOutputs(java.lang.Double[] x)<br>Uses calculateY_i(Double) to retrieve the final y_i value for each neuron of the network. |
| void | calculatePreSynapticFields(java.lang.Double[] x)<br>Calculates the pre-synaptic field for an input pattern using the previouslycalculated weights in calculateWeights(Double[]). |
| abstract void | calculateThresholds(java.lang.Double[] currentPattern, java.lang.Boolean adaptiveFlag)<br>Calculates the Neuron thresholds using either the presynaptic fields of the network h, either the neurons status y or either the data from an input pattern. |
| protected abstract java.lang.Double | calculateW_ij(java.lang.Double x_i, java.lang.Double x_j)<br>Calculates a weight in a NeuralNetwork according to Hebb's law for supervised ANNs |
| void | calculateWeights(java.lang.Double[] x)<br>Updates the weights values for each layer when a new Pattern is learned. |
| protected abstract java.lang.Double | calculateY_i(java.lang.Double h_i)<br>Calculates the post-synaptic potential for a HopfieldNetwork instance |
| java.lang.Double | getC()<br>Getter |
| java.lang.Integer | getK()<br>Getter |
| java.lang.Integer | getM()<br>Getter |
| java.lang.Double[] | getNeurons()<br>Getter |
| java.lang.Double[] | getPreSynapticFields()<br>Getter |
| NeuralUtils | getUtils()<br>Getter |
| void | incrementM()<br>Increments the value of m (number of learned Patterns) |
| private void | initArrays()<br>Initializes the object arrays (weights and inputs). |
| protected abstract NeuralUtils | initUtils()<br>Initializes the associated Utils class of the network depending on the current instance of HopfieldNetwork |
| java.lang.String | printNeurons()<br>Prints the status of the Neurons of a HopfieldNetwork into the a String object. |
| private java.lang.String | printValue(java.lang.Double value)<br>Returns a string representation for a Neuron value |
| protected void | setTopology(java.lang.Integer i)<br>Sets the topology parameters used by this network for this test. |
| void | setU(java.lang.Double u)<br>Setter |
| java.lang.String | toString()<br>toString |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.input

## Class InputData

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.input.InputData

---

```
public class InputData
extends java.lang.Object
```

InputData class This object is returned by InputDataReader after an input file has been correctly read. It is composed of a TreeMap of Patterns, `patterns` and some relevant data parameters: `p` Total number of patterns read `n` Total number of attributes per Pattern (rows x columns) `rows` Rows per Pattern (number of Attributes) `columns` Columns per attribute (number of Values per Attribute) network.

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`InputDataReader`, `Pattern`, `Attribute`

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| private java.lang.Integer | columns |
| private java.lang.Integer | n |
| private java.lang.Integer | p |
| private java.util.TreeMap<java.lang.String,Pattern> | patterns |
| private java.lang.Boolean | printMatrix |
| private java.lang.Integer | rows |
| private InputType | type |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| InputData(InputType type, java.lang.Integer n_0, java.lang.Integer w_0, java.lang.Boolean printMatrix)<br>Class constructor. |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| void | addPattern(Pattern p)<br>Adds a Pattern into the Pattern array and increments by one the number of contained Patterns (p) if the pattern is new. |
| private java.lang.Integer | calculateN()<br>Calculates the total number of attributes (n) |
| java.lang.Integer | getColumnCount()<br>Getter |
| InputType | getDataType()<br>Getter |
| java.lang.Integer | getN()<br>Getter |
| java.lang.Integer | getP()<br>Getter |
| Pattern | getPattern(java.lang.String id)<br>Returns the pattern with the given ID in the Pattern array |
| java.util.Collection<Pattern> | getPatterns()<br>Returns the pattern array for this InputData object |
| java.lang.Integer | getRowCount()<br>Getter |
| void | setN_0(java.lang.Integer n_0)<br>Setter |
| java.lang.Double[][][] | toArray()<br>toArray |
| java.lang.String | toString()<br>toString |

es.uam.eps.tfg.corruptionTNN.common.exception

## Class InputDataException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      es.uam.eps.tfg.corruptionTNN.common.exception.TestException
        es.uam.eps.tfg.corruptionTNN.common.exception.InputDataException

**All Implemented Interfaces:**

java.io.Serializable

---

```
public class InputDataException
extends TestException
```

InputDataException class The program throws this exception when the reading process of an input file fails due to bad formatting/corruption or trying to read unsupported data types (detailed in Globals)

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

Serialized Form

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| private static long | serialVersionUID<br>Generated serial version for Serializable Objects |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| InputDataException(java.lang.String string)<br>Class Constructor |

### Method Summary

**Methods inherited from class java.lang.Throwable**

addSuppressed, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, getSuppressed, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.input

## Class InputDataReader

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.input.InputDataReader

---

```
public class InputDataReader
extends java.lang.Object
```

InputDataReader class. This class creates an InputData object after reading a file pointed by the member `inputFile`: 1. It firstly reads the number of Patterns by detecting how many years is there data of. 2. After that it retrieves each unique country using the first token of each row (line). 3. Then measures the maximum number of columns parsing again every line. 4. Finally, it creates the InputData object and populates it with the data provided by the input file and returns the object.

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`InputData`, `InputType`

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| `private java.io.File` | `inputFile` |
| `private java.lang.Boolean` | `printMatrix` |
| `private static java.lang.String` | `READ_ATTRIBUTES`<br>Read mode flag for reading a row of values - Attribute |
| `private static java.lang.String` | `READ_PATTERNS`<br>Read mode flag for reading a matrix of values - Pattern |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| `InputDataReader(java.io.File f, java.lang.Boolean printMatrix)`<br>Class constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| `private InputData` | `createNewDataSet(InputType dataType, java.lang.Boolean printMatrix)`<br>Creates a new InputData structure, detecting the unique number of Patterns (matrixes) and Attributes (rows) to be read. |
| `InputData` | `readInputData(InputType dataType)`<br>Core method. |
| `private InputData` | `readInputsIntoDataSet(InputData id)`<br>Populates the previously created InputData structure with the input file data by parsing it again and storing each value in the previously generated structures for this object. |
| `private java.util.HashSet<java.lang.String>` | `readItemSet(java.lang.String code)`<br>Reads each unique Pattern or Attribute identifier into a String set. |
| `private java.lang.Integer` | `readMaxValuesPerLine()`<br>Reads the maximum number of columns the input data file has, so the value w0 can be retrieved and applied for the creation of a new InputData object |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.common.types

## Enum InputType

java.lang.Object
  java.lang.Enum<InputType>
    es.uam.eps.tfg.corruptionTNN.common.types.InputType

**All Implemented Interfaces:**

java.io.Serializable, java.lang.Comparable<InputType>

---

```
public enum InputType
extends java.lang.Enum<InputType>
```

InputDataType enumeration Supported input data types

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

### Enum Constant Summary

Enum Constants

| Enum Constant and Description |
|---|
| BGC |
| BGCIPC |
| GENERIC |
| IPC |
| RANDOM |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| static InputType | valueOf(java.lang.String name)<br>Returns the enum constant of this type with the specified name. |
| static InputType[] | values()<br>Returns an array containing the constants of this enum type, in the order they are declared. |

### Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

### Methods inherited from class java.lang.Object

getClass, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.common.utils

## Class InputUtils

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.common.utils.InputUtils

---

```
public class InputUtils
extends java.lang.Object
```

InputUtils class Contains methods for the construction of an InputData object by the class InputDataReader

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `InputUtils()` |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `java.lang.Boolean` | `isACountry(java.lang.String buffer)`<br>Checks if a token read by InputDataReader corresponds to a country through a regular expression [string containing only letters] |
| `java.lang.Boolean` | `isAYear(java.lang.String buffer)`<br>Checks if a token read by InputDataReader corresponds to a year through a regular expression [number of 4 digits] |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN

## Class Main

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.Main

---

```
public class Main
extends java.lang.Object
```

Main class.

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

### Constructor Summary

Constructors

| Modifier | Constructor and Description |
|---|---|
| private | Main() |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| static void | main(java.lang.String[] args)<br>Launches the program, catches and manages the exceptions |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.common.utils

## Class MathUtils

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.common.utils.MathUtils

---

```
public class MathUtils
extends java.lang.Object
```

MathUtils class Contains methods for doing generic statistic and mathematical operations

**Since:**

1.3

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`NeuralUtils`, `TestUtils`

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `MathUtils()` |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `java.lang.Double` | `average(java.lang.Double[] ns)`<br>Calculates the average for a 3-dimensional array (usually a formatted InputData object) iterating over every value in it. |
| `java.lang.Double` | `averageNZ(java.lang.Double[] ns)`<br>Calculates the average for an input array without counting the zero values. |
| `java.lang.Boolean` | `coinDrop()`<br>Returns a randomly generated Boolean value. |
| `java.lang.Double` | `sigmoid(java.lang.Double t)`<br>Sigmoid function |
| `java.lang.Double` | `stdDeviation(java.lang.Double[] ns)`<br>Calculates the standard deviation for a 3-dimensional array (usually a formatted InputData object) iterating over every value in it. |
| `java.lang.Double` | `variance(java.lang.Double[] ns)`<br>Calculates the variance for a 3-dimensional array (usually a formatted InputData object) iterating over every value in it. |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.common.exception

## Class NeuralException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      es.uam.eps.tfg.corruptionTNN.common.exception.TestException
        es.uam.eps.tfg.corruptionTNN.common.exception.NeuralException

**All Implemented Interfaces:**

java.io.Serializable

---

```
public class NeuralException
extends TestException
```

NeuralException class The program throws this exception when the setting of a HopfieldNetwork do not comply the set specifications (unused in this version).

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

Serialized Form

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| private static long | serialVersionUID<br>Generated serial version for Serializable Objects |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| NeuralException(java.lang.String string)<br>Class Constructor |

### Method Summary

#### Methods inherited from class java.lang.Throwable

addSuppressed, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, getSuppressed, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.common.utils

## Class NeuralUtils

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils

**Direct Known Subclasses:**

BimodalNeuralUtils, BinaryNeuralUtils, TernaryNeuralUtils

---

```
public abstract class NeuralUtils
extends java.lang.Object
```

NeuralUtils class Contains methods for threshold adjustment, noise generation and outputs, presynaptic fields and weights calculation, for a NeuralNetwork object. Most of the methods in this class are abstract, as they work differently depending on the employed network type

**Since:**

1.3

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

ArrayUtils

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| NeuralUtils() |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| java.lang.Integer | activity(java.lang.Double[] pattern)<br>Calculates the number of non-null values in a pattern or a neuron array |
| java.lang.Double | activityRatio(java.lang.Double[] pattern)<br>Calculates the ratio of non-null values in a pattern or a neuron array |
| java.lang.Double[][] | convertToNeuralFormat(java.lang.Double[][] inputArray)<br>Converts the internal values of a 3-dimensional array (usually a formatted InputData object) into sigmoid values (-1...1), according to the indicated HopfieldNetwork |
| abstract java.lang.Double | convertToY_i(java.lang.Double x, java.lang.Double u, java.lang.Double sm, java.lang.Double sp)<br>Converts a value into a valid value for the neurons of a HopfieldNetwork |
| java.lang.Double[][] | extendPatternArray(java.lang.Double[][] patterns, java.lang.Integer k)<br>Extends an array of patterns by adding a set of randomly generated patterns at the end of the original pattern array. |
| abstract java.lang.Double | generateNoise()<br>Randomly returns a valid value for the provided neural context in order to generate noise in a Pattern |
| java.lang.Double[] | generateNoisyPattern(java.lang.Double[] pattern, java.lang.Double percentage)<br>Randomly modifies the internal values of a pattern array according to the indicated neural context and a defined amount of noise |
| java.lang.Double[][] | generateRandomPatterns(java.lang.Integer p, java.lang.Integer n)<br>Generates an array of uniform randomly generated patterns inside the neural context provided. |
| abstract java.lang.Double | invertValue(java.lang.Double ref)<br>Inverts the reference value inside the provided context |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.output

## Class OutputData

java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
            org.jfree.ui.ApplicationFrame
              es.uam.eps.tfg.corruptionTNN.output.OutputData

**All Implemented Interfaces:**

java.awt.event.WindowListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class OutputData
extends org.jfree.ui.ApplicationFrame
```

OutputData class. This class creates an OutputData using the JFreeChart library, in order to present the retrieved overlap results by the Test class in a graph format. It creates the data series during the execution of the test and populates `m` and `l` as it calculates the result of these two parameters for each pattern. After the test ends, the resultant graph is constructed.

**Since:**

2.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

Serialized Form

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| private org.jfree.data.xy.XYSeries | l |
| private org.jfree.data.xy.XYSeries | m |
| private java.lang.Boolean | plotCompleteXAxis |
| private static long | serialVersionUID |
| private java.lang.Boolean | softenFlag |
| private java.lang.Double | softFactor |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| OutputData(java.lang.String name, java.lang.Double d, java.lang.Double n, java.lang.Double c, java.lang.Boolean extendFlag, java.lang.Boolean softenFlag, java.lang.Double softFactor)<br>Class constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| void | addPointL(java.lang.Double a, java.lang.Double l)<br>Adds a point representing bimodal overlap when the current instance of the neural network has a pattern load of "a" |
| void | addPointM(java.lang.Double a, java.lang.Double m)<br>Adds a point representing binary overlap when the current instance of the neural network has a pattern load of "a". |
| void | create(java.lang.Integer patternCount)<br>Constructs and displays the resulting graph after the test has ended using `m` and `l` as datasets. |
| void | softenDataSeries(org.jfree.data.xy.XYSeries data)<br>Softens a XYSeries object by removing those consecutive values that exceed the standard deviation of the sample. |

### Methods inherited from class org.jfree.ui.ApplicationFrame

windowActivated, windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified, windowOpened

es.uam.eps.tfg.corruptionTNN.input

## Class Pattern

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.input.Pattern

---

```
class Pattern
extends java.lang.Object
```

Pattern class This object represents a matrix of values for an input file (includes rows and columns), which are stored into the `attributes` member. It also contains an ID that is assigned at the moment of the construction `patternId`

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`Pattern`, `InputData`

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| `private java.util.TreeMap<java.lang.String,Attribute>` | `attributes` |
| `private java.lang.String` | `patternId` |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| `Pattern(java.lang.String patternId)`<br>Class Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| `void` | `addAttribute(Attribute a)`<br>Adds an Attribute into the Attributes map (uses its ID as key) |
| `Attribute` | `getAttribute(java.lang.String id)`<br>Returns the Attribute with the given ID in the Attribute map |
| `java.lang.Integer` | `getAttributeCount()`<br>Getter |
| `Attribute` | `getFirstAttribute()`<br>Returns the first Attribute in the Attribute TreeMap (uses alphabetical order to retrieve it) |
| `java.lang.Integer` | `getInputCount(java.lang.Integer i)`<br>**Deprecated.**<br>*This method is inefficient and should not be used anymore.* |
| `java.lang.String` | `getPatternId()`<br>Returns the given ID for this Pattern object |
| `java.lang.Integer` | `getTotalValueCount()`<br>Returns the total number of values summing up each count for each stored Attribute in this pattern |
| `java.lang.Double[][]` | `toArray()`<br>toArray |
| `java.lang.String` | `toString()`<br>toString |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.common.utils

## Class TernaryNeuralUtils

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils
    es.uam.eps.tfg.corruptionTNN.common.utils.TernaryNeuralUtils

---

```
public class TernaryNeuralUtils
extends NeuralUtils
```

TernaryNeuralUtils class Contains implementations for each abstract method of NeuralUtils using the context of a ternary Hopfield network. Combines utilities of BinaryNeuralUtils and BimodalNeuralUtils in order to create sigma version of a Pattern or a Neuron array

**Since:**

1.3

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

NeuralUtils, BinaryNeuralUtils, BimodalNeuralUtils

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| TernaryNeuralUtils() |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| java.lang.Double[][] | convertToNeuralFormat(java.lang.Double[][] neuralArray)<br>Converts the internal values of a 3-dimensional array (usually a formatted InputData object) into sigmoid values (-1...1). |
| java.lang.Double | convertToY_i(java.lang.Double x, java.lang.Double u, java.lang.Double sm, java.lang.Double sp)<br>Converts a value into a valid value for the neurons of a HopfieldNetwork |
| java.lang.Double | generateNoise()<br>Randomly returns a valid value for a ternary context in order to generate noise in a Pattern |
| java.lang.Double[] | generateSigma(java.lang.Double[] pattern)<br>Converts an array with ternary values into a binary one (generates the Sigma function for a pattern). |
| java.lang.Double[] | generateSigma2(java.lang.Double[] pattern)<br>Generates the SigmaÂ² function for a pattern (squares each value of the array and saves it into a new one, which is returned) |
| java.lang.Double | invertValue(java.lang.Double ref)<br>Inverts the reference value using a ternary context |

### Methods inherited from class es.uam.eps.tfg.corruptionTNN.common.utils.NeuralUtils

activity, activityRatio, extendPatternArray, generateNoisyPattern, generateRandomPatterns

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.test

## Class TernaryTest

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.test.Test
    es.uam.eps.tfg.corruptionTNN.test.TernaryTest

---

```
class TernaryTest
extends Test
```

### Field Summary

**Fields inherited from class es.uam.eps.tfg.corruptionTNN.test.Test**

adaptiveThresholds, bmn, bnn, data, extendedMode, maxA, maxIterations, name, noiseAmmount, output, printPatterns, printSimple, randomPatterns

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| TernaryTest(java.lang.String name, InputType inputType, java.lang.String filePath, java.lang.Boolean extendedMode, java.lang.Double noiseAmmount, java.lang.Integer iterations, java.lang.Double nFraction, java.lang.Double biquadConstant, java.lang.Boolean adaptiveThresholds, java.lang.Boolean printSimple, java.lang.Boolean printMatrix, java.lang.Boolean printPatterns, java.lang.Boolean printGraph, java.lang.Boolean softenGraphFlag, java.lang.Double softenGraphFactor)<br>Constructor |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| protected void | learningPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the learning phase: 1.a. |
| protected java.lang.String | printPerformanceOverlaps(java.lang.Double m, java.lang.Double l)<br>Prints the overlap values obtained for the current iteration |
| protected java.lang.String | printPerformanceSimple(java.lang.Double a, java.lang.Double m, java.lang.Double l)<br>Prints the performance values for the current iteration in a format compatible with the simple results matrix |
| protected void | recoveryPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the recovery phase: 1. |
| protected java.lang.String | toStringBiquadraticConstant()<br>Prints the information about the biquadratic constant during this session. |
| protected java.lang.String | toStringSimpleMatrix()<br>Prints the headers for the simple results matrix depending on the type of Test instanced during this session. |

**Methods inherited from class es.uam.eps.tfg.corruptionTNN.test.Test**

execute, printPerformance, recoverPattern, toString

**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.test

## Class Test

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.test.Test

**Direct Known Subclasses:**

BimodalTest, BinaryTest, TernaryTest

---

```
public abstract class Test
extends java.lang.Object
```

TestModule class This class is the core of this program. Creates and executes a single test provided by the test batch following the sequence established in the method `execute()`. Test module is composed of the following objects, which are initialized at the constructor: `data` InputData to process `bnn` BinaryHopfieldNetwork instance to use (if applicable) `bmn` BimodalHopfieldNetwork instance to use (if applicable) `output` OutputData object to condense performance results into a graph And the following parameters: `name` TestType name `noiseAmmount` Amount of noise to use during the recovery phases (default value at `Globals.TEST_DEFAULT_NOISE_AMMOUNT`) `maxIterations` Maximum number of iterations to perform before ending the recovery phase of the program (default value at `Globals.TEST_DEFAULT_MAX_ITERATIONS`) `maxA` Maximum pattern load the HopfieldNetwork objects will support during this session `printSimple` Flag to indicate the program to only print the results `printPatterns` Flag to indicate if the Neurons will be printed during this session. `randomPatterns` Flag to indicate the program not to use the InputData object, but to generate random patterns with the method `NeuralUtils.generateRandomPatterns(Integer, Integer)`

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

`ArrayUtils`, `InputDataReader`, `BinaryHopfieldNetwork`, `BimodalHopfieldNetwork`, `TernaryNeuralUtils`, `TestUtils`, `TestType`

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| protected java.lang.Boolean | adaptiveThresholds |
| protected BimodalHopfieldNetwork | bmn |
| protected BinaryHopfieldNetwork | bnn |
| protected InputData | data |
| protected java.lang.Boolean | extendedMode |
| protected java.lang.Double | maxA |
| protected java.lang.Integer | maxIterations |
| protected java.lang.String | name |
| protected java.lang.Double | noiseAmmount |
| protected OutputData | output |
| protected java.lang.Boolean | printPatterns |
| protected java.lang.Boolean | printSimple |
| protected java.lang.Boolean | randomPatterns |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| Test(java.lang.String name, InputType inputType, java.lang.String filePath, java.lang.Boolean extendedMode, java.lang.Double noiseAmmount, java.lang.Integer iterations, java.lang.Double nFraction, java.lang.Double biquadConstant, java.lang.Boolean adaptiveThresholds, java.lang.Boolean printSimple, java.lang.Boolean printMatrix, java.lang.Boolean printPatterns, java.lang.Boolean printGraph, java.lang.Boolean softenGraphFlag, java.lang.Double softenGraphFactor)<br>Constructor |

## Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| void | execute()<br>Executes the test: 1. |
| protected abstract void | learningPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the learning phase: 1. |
| protected java.lang.String | printPerformance(java.lang.Double a, java.lang.Double m, java.lang.Double l)<br>Returns a matrix result with a relation of the network pattern load (a) and the binary and bimodal overlap (where aplicable) |
| protected abstract java.lang.String | printPerformanceOverlaps(java.lang.Double m, java.lang.Double l)<br>Prints the overlap values obtained for the current iteration |
| protected abstract java.lang.String | printPerformanceSimple(java.lang.Double a, java.lang.Double m, java.lang.Double l)<br>Prints the performance values for the current iteration in a format compatible with the simple results matrix |
| protected java.lang.Double[] | recoverPattern(java.lang.Double[] noisyPattern, HopfieldNetwork nn)<br>Tries to recover a previously learned pattern from a noisy input. |
| protected abstract void | recoveryPhase(java.lang.Double[] sigma, java.lang.Double[] sigma2)<br>This method represents the steps to take at the recovery phase: 1. |
| java.lang.String | toString()<br>toString |
| protected abstract java.lang.String | toStringBiquadraticConstant()<br>Prints the information about the biquadratic constant during this session. |
| protected abstract java.lang.String | toStringSimpleMatrix()<br>Prints the headers for the simple results matrix depending on the type of Test instanced during this session. |

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.common.exception

## Class TestException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      es.uam.eps.tfg.corruptionTNN.common.exception.TestException

**All Implemented Interfaces:**

java.io.Serializable

**Direct Known Subclasses:**

InputDataException, NeuralException

---

```
public class TestException
extends java.lang.Exception
```

TestException class The program throws this exception when the InputData specifications exceeds the maximum p/k ratio for a HopfieldNetwork.

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

**See Also:**

Serialized Form

### Field Summary

Fields

| Modifier and Type | Field and Description |
|---|---|
| private static long | serialVersionUID<br>Generated serial version for Serializable Objects |

### Constructor Summary

Constructors

| Constructor and Description |
|---|
| TestException(java.lang.String string)<br>Class Constructor |

### Method Summary

#### Methods inherited from class java.lang.Throwable

addSuppressed, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, getSuppressed, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.test

## Class TestFactory

java.lang.Object
    es.uam.eps.tfg.corruptionTNN.test.TestFactory

---

```
public class TestFactory
extends java.lang.Object
```

TestFactory class Reads the program parameters to create and configure a test based on them. The parameter format can be seen on `Globals.OUTPUT_MANUAL`.

**Version:**

1.4

**Author:**

Guillermo Jerez

**See Also:**

`Test`

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `TestFactory()` |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `Test` | `createTest(java.lang.String[] args)`<br>Test constructor. |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

es.uam.eps.tfg.corruptionTNN.common.types

## Enum TestType

java.lang.Object
  java.lang.Enum<TestType>
    es.uam.eps.tfg.corruptionTNN.common.types.TestType

**All Implemented Interfaces:**

java.io.Serializable, java.lang.Comparable<TestType>

---

```
public enum TestType
extends java.lang.Enum<TestType>
```

TestType enumeration Supported test types

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

### Enum Constant Summary

Enum Constants

| Enum Constant and Description |
|---|
| BIMODAL |
| BINARY |
| TERNARY |

### Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| static TestType | valueOf(java.lang.String name)<br>Returns the enum constant of this type with the specified name. |
| static TestType[] | values()<br>Returns an array containing the constants of this enum type, in the order they are declared. |

### Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

### Methods inherited from class java.lang.Object

getClass, notify, notifyAll, wait, wait, wait

es.uam.eps.tfg.corruptionTNN.common.utils

## Class TestUtils

java.lang.Object
  es.uam.eps.tfg.corruptionTNN.common.utils.TestUtils

---

```
public class TestUtils
extends java.lang.Object
```

TestUtils class Contains methods for performance measurement, used by the Test class. Particularly, this class calculates the values of a (Pattern load), m/l (Pattern overlap) and Pattern Activity.

**Since:**

1.0

**Version:**

2.0

**Author:**

Guillermo Jerez

### Constructor Summary

Constructors

| Constructor and Description |
| --- |
| `TestUtils()` |

### Method Summary

Methods

| Modifier and Type | Method and Description |
| --- | --- |
| `java.lang.Double` | `calculateA(java.lang.Integer p, java.lang.Integer k)`<br>Calculates the pattern load using the number of currently learned patterns and the total number of attributes per pattern (connections) |
| `java.lang.Double` | `calculateBimodalOverlap(java.lang.Double[] x, java.lang.Double[] y)`<br>Calculates the overlap after a recovery phase is ended using the initial Pattern and the neurons state after passing it through the network an initially set number of iterations for a bimodal context |
| `java.lang.Double` | `calculateBinaryOverlap(java.lang.Double[] x, java.lang.Double[] y)`<br>Calculates the overlap after a recovery phase is ended using the initial Pattern and the neurons state after passing it through the network an initially set number of iterations for a binary context |

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`